Similarity-Aware Kanerva Coding for On-Line Reinforcement Learning

Wei Li  
Northeastern University  
360 Huntington Ave  
Boston, Massachusetts 02115, USA  
li.wei@husky.neu.edu

Waleed Meleis  
Northeastern University  
360 Huntington Ave  
Boston, Massachusetts 02115, USA  
meleis@ece.neu.edu

ABSTRACT

A major challenge in reinforcement learning (RL) is use of a tabular representation to represent learned policies with a large number of states or state-action pairs. Function approximation is a promising tool to overcome this deficiency. This approach uses parameterized functions instead of a table to represent learned knowledge and enables generalization. However, existing schemes cannot solve realistic RL problems, with their rapidly increasing demands for approximating accuracy and efficiency.

In this paper, we extend the architecture of Sparse Distributed Memories (SDMs) and propose a novel on-line methodology, similarity-aware Kanerva coding (SAK), that closely represents the learned knowledge for very large-scale problems with significantly fewer parameterized components. SAK directly measures the state variables’ real distances in all dimensions and reformulates a new state similarity metric with an improved definition of state closeness. As a result, our scheme accurately distributes and generalizes knowledge among related states. We further enhance SAK’s efficiency by allowing a limited number of prototype states that have certain similarities to be activated for value approximation so that the risk of over-generalization is hindered. In addition, SAK eliminates size tuning and prototype reallocation for the prototype set, resulting in not only broadened scalability but also significant savings in the amount of necessary prototypes and computational overhead needed for RL. Our extensive experimental results show that SAK achieves more than 48% improvements over existing schemes in learning quality, and reveal that SAK is able to consistently learn good policies for RL with small overhead and short training times, even given roughly tuned scheme parameters.

CCS Concepts

•Computing methodologies → Reinforcement learning; Sequential decision making;

Keywords

Reinforcement learning; function approximation; Kanerva coding; state similarity.

1. INTRODUCTION

RL algorithms have shown great success in solving MDP problems [1, 2, 3]. However, when problems have very large, continuous, and/or multi-dimensional state spaces, developing optimal action policies can be challenging due to (1) the large number of states or state-action pairs that have to be considered when representing the policies; (2) the inability to generalize knowledge; and (3) the long training time needed to learn an effective policy.

Function approximation is the primary tool that current RL agents use to improve learning performance when encountering problems with large-scale state spaces [4]. This technique represents a large state space by a compact form using relative small number of parametrized functions. However, many existing function approximation algorithms require either expert partitioning of the state spaces or complex heuristics to dynamically explore and split promising areas. Furthermore, these approaches do not scale well when the number of state space dimensions increase. The architecture of Sparse Distributed Memories (SDMs) (also called Kanerva coding) [5] has the advantage that an increase of the number of state space dimensions does not exponentially increase the number of needed prototypes, thus providing much better scalability than other existing function approximation techniques. Kanerva coding has been shown to outperform other approaches when applied to large-scale state spaces for real-world applications [6, 7, 8].

Kanerva coding [9] uses a set of prototype states (referred to as prototypes) to represent the input sample state. In practice, the critical problem for Kanerva coding, as well as for other function approximation techniques, is the number of states or state-action pairs it tries to represent is much larger than the number of prototypes it can use. As a result, its approximation ability is largely limited by the quality of the set of prototypes. Our goal is to optimize a set of prototypes’ ability to represent a state space for large-scale optimization problems with low overhead.

In this paper, we propose SAK, a novel on-line algorithm that closely represents the learned knowledge for very large-scale MDP problems with a far small amount of allocated resources than ones needed by competing approaches SAK extends the principle of Sparse Distributed Memories (SDMs) and finds a satisfactory representation for value functions of encountered state-action pairs. Although simple in design, SAK shows its superior performance in terms of both accuracy and approximation efficiency. Specifically, SAK does not need any deletion or addition operation on selective components in the given prototype set. Instead of using
perceived measurements, state, action and reward, are presented as follows:

- **State** $s_t = (v_1, \ldots, v_n)$: a vector of input sensory signals perceived from environment
- **Action** $a_t \in \mathcal{A}(s_t)$: the behavior that can be chosen and executed by agent at state $s_t$
- **Reward** $r_t$: a scalar value received after taking action $a_t$ in state $s_t$, indicating the preference or desirability of taking given state-action pair

In RL, the training process is executed over a number of discrete time steps $t$ during which the agent interacts with the environment and learns its policy. At each time step $t$, based on current policy $\pi$ (a mapping from state space to action space), the RL agent chooses an action $a_t$ from $\mathcal{A}$ in state $s_t$, followed by a transition to new state $s_{t+1}$ and an immediate reward $r_{t+1}$ perceived from environment. The objective of RL agent is to maximize the expected return $R_t$ (the sum of cumulative discounted rewards that the agent receives in the long run).

For a control problem in RL, the most important element is the action-value function $Q^*(s_t, a_t)$ whose value is the expected total amount of rewards the agent accumulates after taking action $a_t$ in state $s_t$ and then continuing behaving under policy $\pi$. The RL agent estimates the optimal action-value $Q^*(s_t, a_t)$ from input training samples. In temporal difference (TD) RL methods [10], an agent uses given input samples to iteratively update estimates of $Q^*(s_t, a_t)$, denoted as $\hat{Q}(s_t, a_t)$, to learn the optimal action-value function $Q^*$.

It is impractical to represent the value functions of states or state-action pairs as separate entries in a table because most practical problems have a large number of states or state-action pairs as a result of continuous and/or high-dimensional state spaces. A function approximator instead uses a vector of $N$ real-valued parameterized components $\mathbf{\hat{\theta}}$ to represent the approximate action-value function $\hat{Q}(s_t, a_t)$ such that the values of $\hat{Q}(s_t, a_t)$ for all state-action pairs drawn from the large state space are mapped into a relatively small number of parameterized components in $\mathbf{\hat{\theta}}$.

$\hat{Q}(s_t, a_t; \mathbf{\hat{\theta}})$, the approximate action-value with respect to parameter vector $\mathbf{\hat{\theta}}$, can be represented by a linear function approximator [9]:

$$\hat{Q}(s_t, a_t; \mathbf{\hat{\theta}}) = \mathbf{\hat{\phi}}^T \mathbf{\hat{\theta}} = \sum_{i=1}^{N} \theta_{a_t(i)} \phi_{s_t,a_t(i)}(\mathbf{\hat{\theta}}) ,$$  \hspace{1cm} (1)

where $\mathbf{\hat{\phi}}$ is a vector of prototype states (referred to as set of prototype) with $N$ components that are constructed by certain state representation scheme, e.g., tile coding, Kanerva coding. When $\phi_{s_t,a_t}$ is a vector with binary prototypes, $\phi_{s_t,a_t}(i)$ is said to be active with respect to state-action pair $(s_t, a_t)$ if $\phi_{s_t,a_t}(i) = 1$ and is said to be inactive otherwise. Based on the linear function approximator, the update rule of $\mathbf{\hat{\theta}}$ is formulated as:

$$\mathbf{\hat{\theta}}_{t+1} \leftarrow \mathbf{\hat{\theta}}_t + \alpha [Q^*(s_t, a_t) - \hat{Q}(s_t, a_t; \mathbf{\hat{\theta}})] \phi_{s_t,a_t} .$$  \hspace{1cm} (2)

### 2.2 State-of-the-art Algorithms

We now describe existing function approximation approaches and their limitations. Tile coding [9] is the dominant function approximation approach and has been widely used to address many practical RL problems. However, tile coding requires expert partitioning of the state space; otherwise many redundant tiles can be generated resulting in wastes of resources and reduced learning speed.

Variants of tile coding have been proposed to optimize partitioning and ignore irrelevant environment information in unimportant state regions, e.g., adaptive tile coding [11] and evolutionary tile coding [12]. The heuristic schemes used by both to decide when and how to split considered state regions not only need to evaluate all information in each dimension but also require related parameters to be carefully tuned. These limitations constraint the effectiveness of both approaches when being applied to complex or large-scale problems.
In tile coding and its variants, even if the partition is appropriately chosen, the number of tiles needed to fully represent the function approximator still exponentially increases with the number of state space dimensions.

2.3 Kanerva Coding

Unlike other function approximation techniques, i.e., tile coding and RBFs [9], Kanerva coding’s representation complexity does not exponentially increase with the state space dimensions. In Kanerva coding, a set of prototype locations (prototypes) \( \mathbb{S} \) is constructed from the state space and then used to represent the entire state space. States are represented by a set of prototypes that can serve as a function approximator to approximate the value functions of states or state-action pairs. As shown in Figure [1], a state \( s \) (or prototype \( p \)) in RL consists of a sequence of state variables, i.e., \( s = (v_1, ..., v_n) \) and \( p = (h_1, ..., h_n) \). If a state variable of \( s \) resides within a radius \( \rho_j \) of the receptive field of the prototype \( p \) with respect to their \( j_{th} \) dimension, e.g., \( \|h_j - v_j\| \leq \rho_j \), prototype variable \( h_j \) is said to be close to state variable \( v_j \) in the \( j_{th} \) dimension.

In Kanerva coding, given a state \( s \) and a state space with dimensionality \( n \), a prototype \( p \) is said to be active or similar with respect to \( s \) if \( p \) is close to \( s \) in enough dimensions. In practice, states in Kanerva coding are considered to be similar as long as they achieve closeness on a required number of dimensions, no matter in how many other dimensions they totally differ. This setting is critical because it enables the complexity of function approximation in Kanerva coding to be unaffected by the increments of the sizes and dimensions of the state space [9].

Kanerva coding typically starts with an initial set of prototypes that are randomly selected from the whole state space. However, the selection of prototypes can have a significant effect on Kanerva coding’s performance, i.e., Kanerva coding’s abilities to represent states and approximate value functions are sensitive to the sizes of receptive fields of all prototypes in use. The inappropriate selected sizes of receptive fields can degrade RL’s learning quality and slow down the training time.

An adaptive adjacency method [13] that adjusts the size of receptive field for each prototype was proposed. According to this approach, the degree of generalization is managed instead of being fixed throughout the entire learning process. However, this approach requires management of the trade-off between eliminating prototype starvation (insufficient number of active prototypes to approximate the action values) and avoiding prototype over-generalization. In practice, no matter how existing approaches extend SDMs, the receptive field is a crucial factor that needs to be carefully considered when designing heuristic strategies and initializing training process.

To eliminate intrinsic issues caused by receptive fields in SDMs, [14] proposed an approach that uses a fuzzy value (a continuous value from 0 to 1 that directly relates to state similarity) to provide a weighted update to the value of each prototype. This approach assumes the state space is binary and the fuzzy value is a function of bit differences between two binary states. However, this assumption does not hold for continuous states, and thus the approach cannot directly solve real-world problems with continuous state spaces. Besides, this approach suffers from over-generalization since each prototype is given a partial update from learned values, no matter how they are unrelated to input training states.

In order to further improve SDM’s performance over existing schemes in terms of approximating accuracy and efficiency, we propose SAK, a variant of SDM that gets rid of the use of receptive field and mitigates parameter tuning overhead. SAK is able to accurately measure state similarities in continuous multi-dimensional state spaces. It succeeds in eliminating prototype starvation and avoiding insufficient resolution. We describe SAK in Sect. 3.

3. ALGORITHM DESIGN

We begin with the introduction of state similarity grade, the most important element in SAK that totally replaces the need of receptive field. Then we describe its employment in SAK and present the heuristic to further enhance SAK.

3.1 State Similarity Grade

To better present the idea of state similarity grade, we first define state-variable distance and state-variable similarity grade. In a continuous state space, a state or prototype is represented by a sequence of state variables and each state variable can be a real continuous value that describe a numeric measurement in one particular dimension.

\[
d_j(s, p) = \frac{|v_j - h_j|}{\text{range}_j/\rho},
\]

\[
m_j(s, p) = e^{-d_j(s, p)},
\]

\[
\mu(s, p) = \min_{j=1,...,a} m_j(s, p).
\]

We define \( d_j(s, p) \), the state-variable distance from input sample state \( s = (v_1, ..., v_n) \) to prototype \( p = (h_1, ..., h_n) \) in dimension \( j \), as the value difference of input state variables in the \( j_{th} \) dimension of state \( s \) and prototype \( p \), normalized by the range of possible values in the \( j_{th} \) dimension divided by a constant factor \( \rho \). See (3). Note that the factor \( \rho \) is a constant value bigger than 1 such that the effect of value difference \( |v_j - h_j| \) on \( d_j(s, p) \) can be multiplicatively amplified. In our experiments, \( \rho \) is set to 10.

We then define the state-variable similarity grade in \( j_{th} \) dimension, denoted as \( m_j \), as a function of \( d_j \) such that the value of \( m_j \) is equal to 1 if the value difference between 2 state variables is 0, and \( m_j \) approaches to 0 if the 2 state variables have far different values. See (4).

Finally, we introduce \( \mu(s, p) \), the state similarity grade of state \( s \) with respect to prototype \( p \). See (5). Its value is defined as the minimal state-variable similarity grade among all dimensions of \( s \) and \( p \). Note that the value of state similarity grade is determined by certain dimension of the 2 states that has the largest state-variable distance.

The state similarity grade is a continuous value in range \([0, 1]\). It can be used to find active prototypes, prototypes that have certain similarities with the input sample states. It also can be used to assign weighted value updates to predict function estimation.

3.2 Similarity-Aware Kanerva Coding

We now describe how the state similarity grade is implemented in SAK. Recall that a function approximator maintains a vector of \( N \) parameter values \( \theta_i \) that corresponds
with \( \tilde{\phi}_{s_t,a_t} \), a set of \( N \) prototypes constructed from the state space. Since \( \tilde{\phi}_{s_t,a_t} \)'s component \( \phi_{s_i,a_i} \) can have a continuous-valued response (in the range of \([0,1]\)) to the encountered state \( s_t \), SAK uses \( \mu(s_t,p_t) \) in place of the binary prototype \( \phi_{s_t,a_t} \) such that:

\[
\hat{Q}(s_t,a_t; \tilde{\theta}_i) = \sum_{i=1}^{\theta_{a_t}(i)} \mu(s_t,p_t) .
\]  

(SAK uses a sample return obtained by Sarsa(0) \([9]\), \(r_{t+1} + \gamma \hat{Q}(s_{t+1},a_{t+1}; \tilde{\theta}_{i+1}) \), to be an estimate of the target \( Q^*(s_t,a_t) \) in (2). After observing one input sample state \( s_t \), SAK updates each component of the parameter vector \( \theta \) at action \( a_t \) with a normalized state similarity grade as follows:

\[
\theta_{a_t}(i) \leftarrow \theta_{a_t}(i) + \alpha [r_{t+1} + \gamma \hat{Q}(s_{t+1},a_{t+1}; \tilde{\theta}_{i+1}) - \hat{Q}(s_t,a_t; \tilde{\theta}_i)] \frac{\mu(s_t,p_t)}{\sum_{k=1}^{N} \mu(s_t,p_k)} .
\]  

**Algorithm 1: Similarity-Aware Kanerva Coding**

| Input: \( \tilde{\theta} \): a set of \( N \) randomly selected prototypes |
| Output: learned \( \theta \)-values |

1. **Procedure Main(\( \tilde{\theta} \))**
2. Initialize \( \tilde{\theta} \) to \( \tilde{\theta} \)
3. for each episode do
4. Initialize \( s \) and \( a \)
5. for each step of episode do
6. Execute action \( a \) in \( s \), observe \( r, s' \)
7. \( \tilde{\mu}, \tilde{\mu}_{\text{activated}} \) = **LoadFeatures**(\( s', \tilde{\theta} \))
8. Calculate approximate value functions with (8)
9. given \( \tilde{\mu}_{\text{activated}} \) and \( \tilde{\mu}_{\text{activated}} \)
10. Choose \( a' \) in \( s' \) by using \( e \)-greedy method
11. Update \( \tilde{\theta} \) with (9) given \( (s,a,r,s',a') \)
12. Set \( s = s' \) and \( a = a' \)
13. Until \( s \) is terminal
14. **Procedure LoadFeatures**(\( s', \tilde{\theta} \))
15. for each prototype \( p_t \) in \( \tilde{\mu} \) do
16. Calculate similarity grade \( \mu_i \) between \( s' \) and \( p_t \) by using (5)–(3)
17. Append \( \mu_i \) to \( \tilde{\mu} \)
18. Mark each component in \( \tilde{\mu} \) as valid
19. for each iteration of \( \text{ActiveN} \) do
20. Iterate through each valid component in \( \tilde{\mu} \) to find the largest component \( \mu_i \)
21. Append \( \mu_i \) to \( \tilde{\mu}_{\text{activated}} \) and append \( p_t \) to \( \tilde{\mu}_{\text{activated}} \)
22. Mark \( \mu_i \) as invalid
23. return \( \tilde{\mu}_{\text{activated}} \) and \( \tilde{\mu}_{\text{activated}} \)

### 3.3 Enhancement for SAK

In this section, we present how SAK is further improved. Equation (6) shows that the approximate action-value of state \( s_t \) and action \( a_t \) is the sum of weighted \( \theta \)-values with respect to all prototypes. In this sense, all prototypes can be seen as being activated by the input sample state \( s_t \). Note that if one visited state \( s_t \) has too many active prototypes, over-generalization can occur, even when real-valued similarity-based \( \mu \) involves in the value estimates and \( \theta \) updates. The reason can be understood intuitively: although each distant prototype \( p_t \) receives only a few updates to its \( \theta \)-value (because corresponding \( \mu(s_t,p_t) \) is very small), since each input training state, among considerably many others, will trigger such updates, the accumulated \( \theta \)-value can be significant enough to affect the value estimates of other state-action pairs.

To avoid over-generalization, we consider two heuristics to select a relatively small set of prototypes to be active. First, we set an upper bound on \( \mu \), referred to as \( \mu_m \), to select qualified prototypes whose \( \mu \) is smaller than \( \mu_m \). Second, we fix the number of active prototypes (referred to as \( \text{ActiveN} \)) that we expect to obtain for each training sample state. To guarantee the existence of \( \text{ActiveN} \) number of active prototypes for each input sample state, we select the first \( \text{ActiveN} \) prototypes with the largest \( \mu \) (i.e., the closest prototypes).

The first heuristic can achieve even better performance if \( \mu_m \) is well selected. However, to mitigate the effects of a poorly selected value for \( \mu_m \), and to minimize the efforts on parameter tunings with the first heuristic, we opt for the second heuristic to be used in SAK. Experimental results showed that the value of \( \text{ActiveN} \) does not significantly affect the learning performance and is therefore easier to tune than parameter \( \mu_m \). A detailed discussion of parameter tunings in SAK is presented in Sect. 4.2.3.

Based on the \( \text{ActiveN} \) heuristic, when estimating the \( Q \)-values and updating the \( \theta \)-values during trainings, SAK only needs to consider the selective set of prototypes \( M \) that contains the first \( \text{ActiveN} \) number of prototypes with the largest \( \mu \). As such, the estimate of the \( Q \)-value with respect to the parameter vector \( \theta \) for SAK is:

\[
\hat{Q}(s_t,a_t; \tilde{\theta}_i) = \sum_{\mu(s_t,p_t)} \mu(s_t,p_t) .
\]

Correspondingly, SAK changes its rule to update each component of the parameter vector \( \theta \):

\[
\theta_{a_t}(i) \leftarrow \theta_{a_t}(i) + \alpha [r_{t+1} + \gamma \hat{Q}(s_{t+1},a_{t+1}; \tilde{\theta}_{i+1}) - \hat{Q}(s_t,a_t; \tilde{\theta}_i)] \frac{\mu(s_t,p_t)}{\sum_{k \in M} \mu(s_t,p_k)} , \forall i \in M .
\]

The pseudocode and implementation details of SAK are shown in Algorithm 1.

### 4. EXPERIMENTAL EVALUATION

We evaluate SAK by applying the technique to a collection of instances in a variant of Hunter-Prey problem \([6]\) that has a more complex, continuous and high-dimensional state space.

#### 4.1 Methodology

In our experiments, we compare SAK with one dominant function approximation algorithm, tile coding (also called CMACs) and the state-of-the-art SDMs approach, adaptive Kanerva coding (denoted as AdaptiveK) that proved to be able to successfully address large-scale RL problems.

We describe above representative baseline algorithms as follows:

- **CMACs**: a function approximator that employs a number of slightly offset tilings each of which exhaustively partitions the entire state space into nonoverlapping tiles. The region of the tile is called the receptive field within which input states are indiscriminate.
- **AdaptiveK**: a SDMs variant that adaptively adjusts the layout of the prototype set through deletion and
addition operations. Prototypes that are rarely visited have high probabilities to be deleted and prototypes that are mostly visited generate new neighboring prototypes to compensate for the lack of prototypes due to deletion.

We evaluate CMACs, AdaptiveK and SAK over Hunter-Prey benchmark domain and compare their performance using averaged results. Since the performance of CMACs is sensitive to the layout of tilings, we present comprehensive results for CMACs with different kinds of partitions and various number of tilings to achieve its best performance. Similarly, our experiments implement AdaptiveK and SAK with different number of prototypes to see how their performance are affected. To make a fair comparison, SAK starts with the same initial set of prototypes as AdaptiveK. We also provide sensitivity analysis on SAK’s parameter ActiveN to understand its effect on approximation performance.

During training, the RL hyper-parameters are configured to be: $\alpha = 0.5$, $\epsilon = 0.0$, and $\gamma = 1.0$. These parameters were experimentally selected and remained fixed throughout our experiments. In all our experimental results except Figure 5, the tunable parameter ActiveN of SAK is set to 10. Varying values of ActiveN in certain range has little effect on SAK’s performance. We present detailed sensitivity analysis in Sect. 4.2.3.

### 4.2 Evaluate SAK with Hunter-Prey

#### 4.2.1 Hunter-Prey Setup

As a highly complex benchmark problem, a variant of Hunter-Prey (see detailed settings in [6]) is used in our experiments. In our Hunter-Prey tasks, one prey has to learn a policy to avoid being captured by two greedy hunters. The full state space and configurations of Hunter-Prey is very complex. The state is specified by 4 continuous variables in which the position of each hunter consists of 2 variables, the radial coordinate and angular coordinate determined by a polar coordinate system, and one extra integer variable indicating the number of living hunters. In our experiments, each episode starts with an initial state in which each hunter is randomly placed on a circle of radius 500 units with the prey as the center.

#### 4.2.2 Comparison to Existing Algorithms

SAK’s superior performance is easily observed in empirical results from the Hunter-Prey problem. The results of applying SAK, AdaptiveK and CMACs to instances of Hunter-Prey are shown in Figure 2. Figure 3 and Figure 4 show more detailed analysis in the form of CDFs and bar charts of results with standard deviation as error bars. SAK outperforms both CMACs and AdaptiveK in terms of learning quality and allocated resources. As demonstrated in Figure 2, SAK with 200 prototype locations converges to a significantly higher return than CMACs (with radii of tiles $(150, 3.14)^2$) and AdaptiveK that use the same amount of prototype locations as SAK, showing improvements of 48.2%–48.7%. In addition, when only using half of the number of prototypes used by AdaptiveK, SAK can also learn better action policies than AdaptiveK, achieving 31.6% improvements in average return.
We observe that CMACs outperforms AdaptiveK when allocating a very large number of prototype locations, i.e., 1024. However, CMACs with 1024 allocated prototype locations still cannot beat SAK that only uses fewer than 10% of prototype locations used by CMACs, leaving a performance gap of 15.3%. Figure 4 shows SAK’s significant savings on the amount of prototype locations used during learning.

4.2.3 Sensitivity Analysis

In this section, we analyze the sensitivity of the performance of SAK to variations in the primary parameter $\textit{ActiveN}$ defined in our algorithm. Recall that $\textit{ActiveN}$ regulates the number of active prototypes only whose $\theta$-values are used to approximate the action-value function of the input state-action pair. See (8). Figure 5 shows the average return and CDFs for SAK that employs various values of $\textit{ActiveN}$, i.e., from 5 to 25. As shown, different values of $\textit{ActiveN}$ in SAK give similarly good learning results. In fact, there is little statistically difference in resulted average returns when $\textit{ActiveN}$’s value varies from 5 to 25. This indicates that SAK does not require significant effort to tune or complex heuristics to explore the best value of $\textit{ActiveN}$ to achieve satisfactory learning results.

5. DISCUSSION

SAK’s simplicity, no need of parameter tuning, and low overhead in terms of allocated resources and training time allow for easy application to new problems with large-scale state spaces. In addition, the new state similarity metric in SAK provides highly accurate weights measurement for value update and function estimation. Thus, the needed number of parameterized components to truly represent the control policies for the full state-action space is significantly reduced. SAK’s these advantages also enable it to learn from a limited amount of input training samples and develop good policies quickly.

Our results suggest that one of the promising realistic applications of SAK can be the deployment in programmable smart devices that have limited resources, such as IoT [15].

6. CONCLUSION

In this paper, we described SAK, a function approximation scheme that represents learned polices for large-scale state spaces with a limited number of parameterized state prototypes. SAK is able to optimize its representation to the function approximator with a reduced number of parameterized prototypes and can shorten the training time as well. Our experiments over the Hunter-Prey benchmark domain confirmed SAK’s advantages over the state-of-the-art function approximators in a broad set of operating configurations. Our empirical results showed that SAK largely outperformed existing function approximation algorithms, revealing 48% improvements in learning quality.

REFERENCES