

Adaptive Kanerva-based Function Approximation for Multi-Agent Systems

(Short Paper)

Cheng Wu and Waleed M. Meleis

ABSTRACT

In this paper, we show how adaptive prototype optimization can be used to improve the performance of function approximation based on Kanerva Coding when solving large-scale instances of classic multi-agent problems. We apply our techniques to the predator-prey pursuit problem. We first demonstrate that Kanerva Coding applied within a reinforcement learner does not give good results. We then describe our new adaptive Kanerva-based function approximation algorithm, based on prototype deletion and generation. We show that probabilistic prototype deletion with random prototype generation increases the fraction of test instances that are solved from 45% to 90%, and that prototype splitting increases that fraction to 94%. We also show that optimizing prototypes reduces the number of prototypes, and therefore the number of features, needed to achieve a 90% solution rate by up to 87%. These results demonstrate that our approach can dramatically improve the quality of the results obtained and reduce the number of prototypes required. We conclude that adaptive prototype optimization can greatly improve a Kanerva-based reinforcement learner's ability to solve large-scale multi-agent problems.

Categories and Subject Descriptors

I.2.8 [Artificial Intelligence]: Problem Solving, Control Methods, and Search

General Terms

Algorithms, Experimentation

Keywords

Function approximation, Kanerva coding, Reinforcement learning, pursuit

1. INTRODUCTION AND RELATED WORK

Multi-agent problems can be difficult to solve by traditional machine learning techniques because the state space can be very large. The predator-prey pursuit problem [4] is a classic example of such a multi-agent problem. A general version of the problem takes place on a rectangular grid with

Cite as: Adaptive Kanerva-based Function Approximation for Multi-Agent Systems (Short Paper), Cheng Wu, Waleed M. Meleis, *Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, Padgham, Parkes, Müller and Parsons (eds.), May, 12-16., 2008, Estoril, Portugal, pp. XXX-XXX.

Copyright © 2008, International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org). All rights reserved.

one or more predator agents and one or more prey agents. Each grid cell is either open or closed, and an agent can only occupy open cells. Each agent has an initial position.

The problem is played in a sequence of time periods. In each time period, each agent can move to a neighboring open cell one horizontal or vertical step from its current location, or it can remain in its current cell. All moves are assumed to occur simultaneously, and more than one predator agent may not occupy the same cell at the same time. Each agent can observe the location of all other agents, and predator agents and prey agents can each communicate with agents of the same type. If a predator agent is in the same cell as a prey agent at the end of a time period, then that target has been caught. The goal is for the predator agents to catch all the prey agents in the shortest time.

Pursuit problems are difficult to solve in general. Closed-form solutions to restricted versions of the problem have been found [1, 7], but most such problems remain open. Researchers have used approaches such as genetic algorithms [5] and reinforcement learning [12] to develop solutions.

Reinforcement learning [11] is, in some respects, well-suited to solving multi-agent problems, and Q-learning [13] has emerged as one of the most successful reinforcement learning strategies. The algorithm works by combining state space exploration and exploitation to learn the value of each state-action pair. Through repeated trials, the estimates of the values of each state-action pair can gradually converge to the true value, and these can be used to guide the agent to maximize its reward. Under certain limited conditions, Q-learning has been shown to converge to an optimal policy.

A key limitation on the effectiveness of Q-learning is the size of the table needed to store the state-action values. The requirement that an estimated value be stored for every state-action pair limits the size and complexity of the learning problems that can be solved. Instead, function approximation [3] can be used to store an approximation of this table. Many approximation techniques exist, including coarse coding [6], and tile coding [2], and there are guarantees on their effectiveness in some cases [11].

Sparse distributed memories [8] can also be used to reduce the amount of memory needed to store the state-action value table. This approach applied to reinforcement learning, also called Kanerva Coding [11], represents a function approximation technique that is particularly well-suited to problem domains with high dimensionality. A collection of k *prototype state-action pairs*, (prototypes) is selected, each of which again corresponds to a binary feature. A state-action pair and a prototype are said to be *adjacent* if their bit-wise

representations differ by no more than 1 bit. A state-action pair is represented as a collection of binary features, each of which equals 1 if and only if the corresponding prototype is adjacent. A value $\theta(i)$ is maintained for the i th feature, and an approximation of the value of a state-action pair is then the sum of the θ values of the adjacent prototypes. In this way, Kanerva Coding can greatly reduce the size of the value table that needs to be stored.

If the number of prototypes is very large relative to the number of state-action pairs, and the prototypes are uniformly distributed through the state space, each prototype will be adjacent to a small number of state-action pairs. In this case, the approximate state-action values will tend to be close to the true values, and the reinforcement learner will operate as usual. However if the number of prototypes is small, or if the prototypes themselves are not well chosen, the approximate values will not be similar to the true values and the reinforcement learner will give poor results.

Adaptively choosing prototypes appropriate to the particular application is an important way to contribute prior knowledge and experience to the reinforcement learner. There is therefore a need for algorithms to select prototypes that can span the state-space for a particular application. There have been few published attempts to apply Kanerva coding to multi-agent problems [9] or to evaluate and improve the quality of sets of prototypes.

Ratitch [10] has shown that sparse distributed memories can be used to represent the value table in a reinforcement learner. However, they add and delete locations only when the number of locations activated by an individual sample is below a fixed threshold. This approach may overreact to individual samples, in contrast to our approach which considers all samples and all prototypes in a training run before adding and deleting locations. Also, the deterministic nature of their decision to delete a prototype is less flexible than our probabilistic approach.

2. PROTOTYPE OPTIMIZATION

When two different state-action pairs visited during Q-learning are mapped to the same subset of the prototypes, a prototype *collision* is said to have taken place. Both state-action pairs will necessarily have the same approximate value, at least one of which may be far from its true value. Selecting a set of prototypes that minimizes collisions will maximize the solver’s ability to solve the problem.

However it is difficult to generate an optimal set of prototypes for several reasons: the space of possible subsets is very large and the state-action pairs encountered by the solver depend on the specific problem instance being solved. We therefore investigate several heuristic solutions to the prototype optimization problem.

We say that a prototype is *visited* during Q-learning if it is adjacent to the current state-action pair. If a specific prototype is rarely visited, it implies that few state-action pairs are adjacent to this prototype. This suggests that this prototype is inappropriate for the particular application. On the contrary, if a specific prototype is visited frequently, it implies that too many state-action pairs are adjacent to the prototype and collisions are more likely to occur. A necessary condition for collisions to be minimized is that most prototypes are visited an average number of times.

The frequency distribution of visits to prototypes over a sample run using Q-learning with Kanerva coding is shown

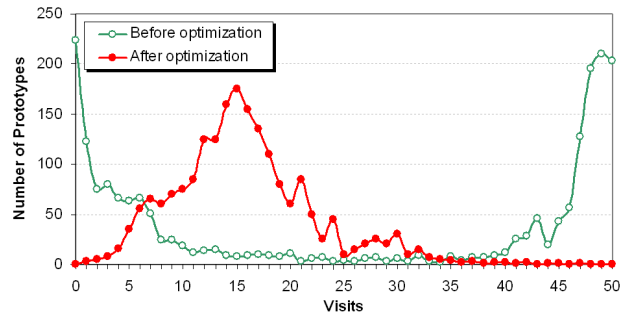


Figure 1: Distribution of the number of visits per prototype in a sample run.

in Figure 1 both before and after prototype optimization. This example is an instance of pursuit with a 32x32 grid, two predator agents, and one prey agent. The non-uniform distribution of visit frequencies across prototypes before prototype optimization indicates that some prototypes are frequently visited and others are rarely visited.

We can optimize prototypes using visit frequencies. We divide the original prototypes into three categories: prototypes with low visit frequency, prototypes with high visit frequency, and the rest of the prototypes. Prototype optimization attempts to replace those prototypes with low or high frequency with prototypes that will have average visit frequencies, as shown in Figure 1.

We describe and evaluate different optimization mechanisms to achieve this goal. In each case, initial prototypes are selected randomly from the entire space of possible state-action pairs. Q-learning with Kanerva coding is used to develop policies for the predator agents, while keeping track of the number of visits to each prototype. After a fixed number of iterations, we update the prototypes using one of the mechanisms described below.

2.1 Prototype deletion

Prototypes that are rarely visited do not contribute to the solution of instances. Similarly, prototypes that are visited frequently are likely to cause many collisions. It makes sense to delete these prototypes and replace them with new prototypes with average frequencies. We evaluate the following two algorithms for deleting prototypes.

In the first approach, we periodically delete a fraction of prototypes whose visit frequency is lowest, and a fraction of prototypes whose visit frequency is highest. The fraction of prototypes that is deleted slowly decreases as the algorithm runs. The θ value and visit frequency of the new prototype is initially set to zero. We refer to this approach as deterministic prototype deletion.

An advantage of this algorithm is that it is easy to implement and it uses application- and instance-specific information to guide the deletion of rarely or heavily visited prototypes. However, this approach deletes prototypes deterministically which does not give the solver the flexibility to keep some prototypes that are rarely or frequently visited. For example, if the number of prototypes is very large, some prototypes that might become useful will not be visited in an early epoch and will be deleted.

In the second approach, we delete prototypes with a probability equal to an exponential function of the number of vis-

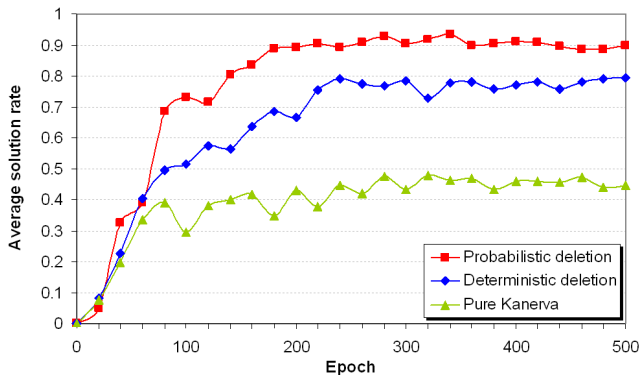


Figure 2: Effect of prototype deletion.

its. I.e. the probability p_{del} of deleting a prototype whose visit frequency is v is $p_{del} = \lambda e^{-\lambda v}$, where λ is a parameter that can vary from 0 to 1. In this approach, prototypes that are rarely visited tend to be deleted with a high probability, while prototypes that are frequently visited are rarely deleted (we describe how we reduce the visit frequency of heavily visited prototypes in the next section). We refer to this approach as probabilistic prototype deletion.

2.2 Prototype generation

We replace prototypes that have been deleted with new prototypes that will tend to improve the behavior of the function approximation. We evaluate the following two algorithms for generating prototypes.

In the first approach, new prototypes are generated randomly from the entire state space. While this approach aggressively searches the state space for useful prototypes, it does not use domain- or instance-specific information.

In the second approach, we create new prototypes by applying prototype splitting. A prototype s_1 that has been visited the most times is selected, and a new prototype s_2 that is a neighbor of s_1 is created by inverting a fixed number of bits in s_1 . The prototype s_1 remains unchanged.

This approach creates new prototypes near prototypes with the highest visit frequencies. These prototypes are similar but distinct which tends to reduce the number of visits to nearby prototypes, and therefore the number of collisions they cause.

3. EXPERIMENTAL EVALUATION

We evaluate our prototype optimization algorithms by applying them to random predator-prey pursuit instances on a 32x32 grid with two non-communicating predator agents and one prey agent. Each predator agent can see the position of the prey agent. Each agent can select one of 9 possible actions, moving one step in any of 8 directions, or not moving. Each grid instance has 32 random closed cells.

In each epoch, we apply each learning algorithm with 1984 prototypes to 40 random training instances followed by 40 random test instances. Prototype optimization is applied after every 20 epochs. For every 20 epochs, we record the average fraction of test instances within those epochs that are solved within a maximum of 64 moves.

The effect of different prototype deletion algorithms is shown in Figure 2. The figure shows the average fraction of test instances solved over a series of epochs for three al-

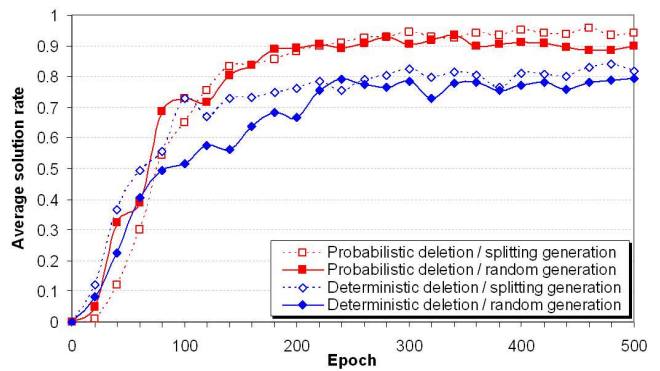


Figure 3: Effect of prototype generation.

gorithms: the pure Kanerva coding algorithm that uses no prototype optimization, deterministic deletion, and probabilistic deletion algorithms. These deletion algorithms use random prototype generation.

The algorithms converge after about 200 epochs, and the results show that the pure Kanerva algorithm solves approximately 45% of the test instances, the deterministic-deletion algorithm solves approximately 79% of the test instances, and the probabilistic-deletion algorithm solves approximately 90% of the test instances. These results indicate that dynamically deleting and regenerating prototypes can significantly increase the quality of the results. The results also indicate that probabilistic prototype deletion significantly outperforms deterministic deletion.

The effect of different prototype generation algorithms is shown in Figure 3. The figure shows the average fraction of test instances solved over a series of epochs for all four combinations of deletion and generation algorithms.

The algorithms converge after about 240 epochs, and the results show that prototype splitting raises the fraction of test instances solved from 79% to 82% with deterministic prototype deletion, and from 90% to 94% with probabilistic prototype deletion. These results indicate that prototype splitting can improve the quality of the results by a small but noticeable amount.

The effect of varying the parameter λ in the exponential distribution used to delete prototypes in the probabilistic deletion algorithm is shown in Table 1. The table shows the average fraction of test instances solved over a range of λ values with either random prototype generation or prototype splitting. The results show that the best results are achieved when $\lambda = 1$ for both prototype generation algorithms.

Figure 4 shows the minimum number of prototypes needed to solve an average of 90% of test instances over a range of grid sizes. The results compare the pure Kanerva algorithm with the probabilistic-split algorithm with $\lambda = 1$. The al-

λ	0	0.5	0.8	1
Random generation	56.25%	77.00%	86.38%	90.40%
Splitting generation	60.51%	82.63%	94.13%	94.25%

Table 1: The effect of λ under probabilistic deletion

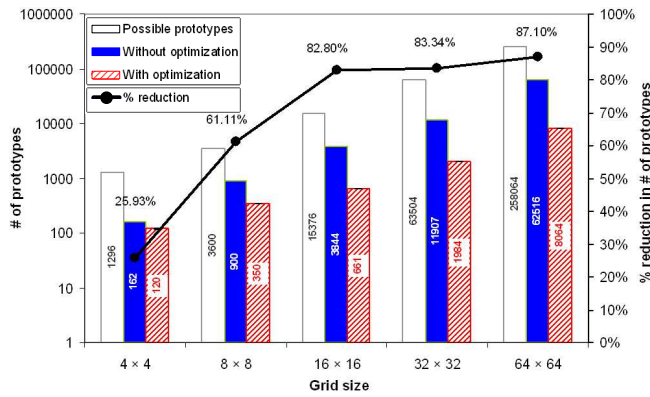


Figure 4: Minimum number of prototypes to solve an average of 90% of instances, and % reduction.

gorithm is run for 500 epochs and the average solution rate is measured over the next epoch. The results are computed by initially setting the number of prototypes equal to the total number of possible prototypes. After 500 epochs, if the result is greater than 90%, the number of prototypes is gradually decreased and the results are recomputed. This process continues until the solution rate is less than 90%. We report the minimum number of prototypes needed to solve an average of 90% of the test instances, which is shown on a logarithmic scale. Figure 4 also shows the total number of possible prototypes and the percent reduction in the number of prototypes needed.

The results show that prototype optimization dramatically reduces the number of prototypes needed to achieve a 90% solution rate. For example, on a 64x64 grid the number of prototypes needed is reduced from 62,516 to 8,064, a reduction of 87%.

We show an example of the policy learned after 500 epochs using our adaptive Kanerva-based function approximation algorithm in Figure 5. This example is an instance of pursuit with a 32x32 grid, one prey agent which starts on the left, and two predator agents.

4. CONCLUSIONS

We have shown that pure Kanerva-based function approximation applied within a reinforcement learner does not give good results. We described our new adaptive Kanerva-based function approximation algorithm, based on prototype deletion and generation. We showed that probabilistic prototype deletion with random prototype generation increases the fraction of test instances that are solved from 45% to 90%, and that prototype splitting increases that fraction to 94%. We also showed that optimizing prototypes reduces the number of prototypes, and therefore the number of features, needed to achieve a 90% solution rate by up to 87%.

These results demonstrate that our approach can dramatically improve the quality of the results obtained and reduce the number of prototypes required. We conclude that adaptive prototype optimization can greatly improve a Kanerva-based reinforcement learner’s ability to solve large-scale multi-agent problems.

5. REFERENCES

[1] M. Adler, H. Racke, N. Sivadasan, C. Sohler, and

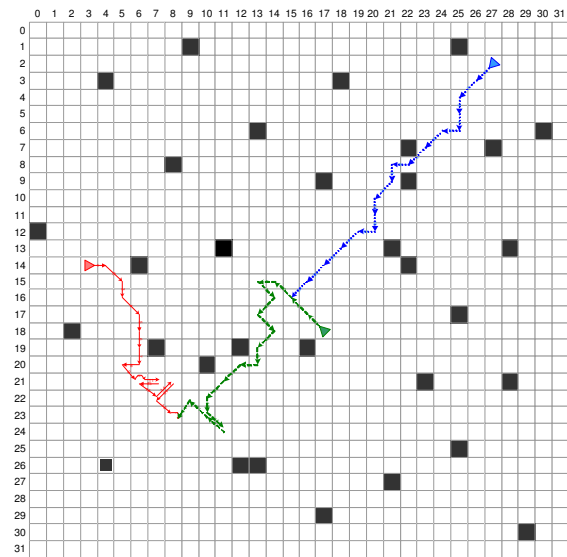


Figure 5: Sample policy

B. Vocking. Randomized pursuit-evasion in graphs. In *Proc. of the Intl. Colloq. on Automata, Languages and Programming*, 2002.

- [2] J. Albus. *Brains, Behaviour, and Robotics*. McGraw-Hill, 1981.
- [3] L. Baird. Residual algorithms: Reinforcement learning with function approximation. In *Proc. of the 12th Intl. Conf. on Machine Learning*. Morgan Kaufmann, 1995.
- [4] M. Benda, V. Jagannathan, and R. Rodhiawalla. On optimal cooperation of knowledge sources. *Technical Report, Boeing Computer Services*, 1985.
- [5] T. Haynes and S. Sen. The evolution of multiagent coordination strategies. *Adaptive Behavior*, 1997.
- [6] G. Hinton. Distributed representations. *Technical Report, Department of Computer Science, Carnegie-Mellon University, Pittsburgh*, 1984.
- [7] V. Isler, S. Kannan, and S. Khanna. Randomized pursuit-evasion with local visibility. *SIAM Journal on Discrete Mathematics*, 20(1):26–41, 2006.
- [8] P. Kanerva. *Sparse Distributed Memory*. MIT Press, 1988.
- [9] K. Kostiadis and H. Hu. KaBaGe-RL: Kanerva-based generalisation and reinforcement learning for possession football. In *Proc. of IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems*, 2001.
- [10] B. Ratitch and D. Precup. Sparse distributed memories for on-line value-based reinforcement learning. In *Proc. of the European Conf. on Machine Learning*, 2004.
- [11] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. Bradford Books, 1998.
- [12] M. Tan. Multi-agent reinforcement learning: Independent vs. cooperative learning. In M. N. Huhns and M. P. Singh, editors, *Readings in Agents*, pages 487–494. Morgan Kaufmann, CA, 1997.
- [13] C. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8:279–292, 1989.