Scaling Model-Based Average-Reward Reinforcement Learning for Product Delivery

Scott Proper[1] and Prasad Tadepalli[2]

[1] Oregon State University, Corvallis, OR 97331-3202, USA,
proper@cs.orst.edu
[2] tadepall@cs.orst.edu

Abstract. Reinforcement learning in real-world domains suffers from three curses of dimensionality: explosions in state and action spaces, and high stochasticity. We present approaches that mitigate each of these curses. To handle the state-space explosion, we introduce "tabular linear functions" that generalize tile-coding and linear value functions. Action space complexity is reduced by replacing complete joint action space search with a form of hill climbing. To deal with high stochasticity, we introduce a new algorithm called ASH-learning, which is an afterstate version of H-Learning. Our extensions make it practical to apply reinforcement learning to a domain of product delivery - an optimization problem that combines inventory control and vehicle routing.

1 Introduction

Reinforcement Learning (RL) provides a nice framework to model a variety of stochastic optimization problems [1]. However, table-based approaches to large RL problems suffer from three "curses of dimensionality": explosions in state and action spaces, and a large number of possible next states of an action due to stochasticity [2]. We propose ways to mitigate these curses in a moderately-sized domain of real-time delivery of products using multiple vehicles with stochastic demands. While RL has been applied separately to inventory control [3] and vehicle routing [4,5,2,6] in the past, we are not aware of any applications of RL to the integrated problem of real-time delivery of products that includes both.

We introduce methods that effectively address each of the curses of dimensionality in the product delivery domain. To mitigate the exploding state-space problem, we introduce "tabular linear functions," (TLFs) which can be viewed as linear functions over some features, whose weights are functions of other features. TLFs generalize tables, linear functions, and tile coding, and allow for a fairly flexible mechanism for specifying the space of potential value functions. We show particular uses of these functions in the product delivery domain that achieve a compact representation of the value function and faster learning.

Second, to reduce the computational cost of searching the action space, which is exponential in the number of agents, we introduce a hill climbing algorithm that scales to a larger number of agents without sacrificing solution quality.

Third, since our base algorithm is model-based, we must calculate the expected value of the next state each step. Many domains have a large stochastic branching factor, i.e., large number of possible next states for a given state-action pair. To mitigate this problem, we introduce ASH-Learning, which is an "afterstate" version of H-learning, and learns by distinguishing between the action-dependent and action-independent effects of the action [1].

In Section 2, we give background on Average-reward Reinforcement Learning (ARL) and H-learning. In Section 3 we illustrate the three curses of dimensionality in the product delivery domain which motivates our research. In Section 4, we describe our solutions to the three curses of dimensionality. In Section 5, we present experimental results and in Section 6 we discuss our results.

2 Average-Reward Reinforcement Learning

We assume that the learner's environment is modeled by a Markov Decision Process (MDP), defined by a 5-tuple $\langle S, A, U, p, r\rangle$, where S is a discrete set of N states, and A is a discrete set of actions. $U(s)$ is the set of actions applicable in state s. By the Markovian assumption, an action u in a given state $s \in S$ results in state s' with some fixed probability $p(s'|s,u)$ and a finite immediate reward $r(s,u)$. A *policy* μ is a mapping from states to actions, such that $\mu(s) \in U(s)$. In Average-reward Reinforcement Learning (ARL), we seek to optimize the average expected reward per time-step, which is called the *gain* [7]. For a given starting state s_0 and policy μ, the gain is given by Equation 1 where $r^{\mu}(s_0,t)$ is the total reward in t steps when policy μ is used starting at state s_0, and $E(r^{\mu}(s_0,t))$ is its expected value:

$$\rho^{\mu}(s_0) = \lim_{t\to\infty} \frac{1}{t} E(r^{\mu}(s_0,t)) \tag{1}$$

The goal of ARL is to learn a policy that achieves near-optimal gain by executing actions, receiving rewards and learning from them. For any fixed policy, the limit of the difference between the total reward accumulated in time t from a state s and the total reward expected in time t on the average as t tends to infinity is called the *bias* of s and is denoted by $h(s)$. For an optimal policy, the gain ρ and the bias $h(.)$ satisfy the following Bellman equation [7]:

$$h(s) = \max_{u\in U(s)} \left\{ r(s,u) + \sum_{s'=1}^{N} p(s'|s,u)h(s') \right\} - \rho \tag{2}$$

The optimal policy chooses actions maximizing the right hand side of this equation. We use an ARL method called "H-Learning" which is model-based in that it learns and uses explicitly represented action models $p(s'|s,u)$ and $r(s,u)$ [8].

At every step, the H-learning algorithm updates the parameters of the value function in the direction of reducing the temporal difference error (TDE), i.e., the difference between the r.h.s. and the l.h.s. of the Bellman Equation 2.

$$TDE(s) = \max_{u\in U(s)} \left\{ r(s,u) + \sum_{s'=1}^{N} p(s'|s,u)h(s') \right\} - \rho - h(s) \tag{3}$$

We use ϵ-greedy exploration in all our experiments. From Equation 2, it can be seen that $r(s,u) + h(s') - h(s)$ gives an unbiased estimate of ρ, when action u is greedy in state s and s' is the next state. Hence, H-Learning updates ρ as follows in every greedy step:

$$\rho \leftarrow (1-\alpha)\rho + \alpha(r(s,a) - h(s) + h(s')) \tag{4}$$

3 The Product Delivery Domain

To conduct our experiments, we used a version of the product delivery domain (shown in Figure 1) that combines aspects of the vehicle routing problem [9] and inventory control problems. A single product is to be delivered to shops from a depot using several trucks. Shop inventories and truck loads are discretized into 5 levels 0-4. We used 4 trucks, 5 shops, and 10 locations, giving a state-space size of $(5^5)(5^4)(10^4) = 19{,}531{,}250{,}000$, which illustrates the first curse of dimensionality. Each truck has 9 actions available at each time step: unload up to 4 units, move in up to 4 directions, or wait. With 4 trucks, we must consider $9^4 = 6561$ joint actions each step, thus illustrating the second curse of dimensionality. Trucks are loaded automatically upon reaching the depot. A small negative reward of -0.1 is given for every "move" action to reflect fuel costs. We give a reward of -5 if a customer enters a store and finds the shelves empty. We model customer consumption at shops by decreasing the inventory level by 1 unit with some probability, which independently varies from shop to shop. Thus the number of possible next states for a state and an action, called the "stochastic branching factor," is exponential in the number of shops, illustrating the third curse of dimensionality.

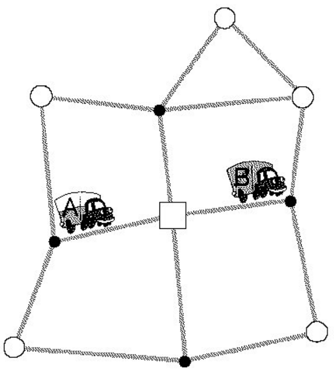

Fig. 1. The Product Delivery Domain: The square in the center is the depot and the circles are the shops

4 Taming the Three Curses of Dimensionality

This section describes our methods for taming the three curses of dimensionality.

4.1 Tabular Linear Functions

We introduce "tabular linear functions," (TLFs) which generalize linear functions, tables, and tile coding. A TLF is a linear function of a set of "linear" features of the state, where the weights of the linear function are arbitrary functions of other discretized or "nominal" features. Hence the weights can be stored in a table indexed by the nominal features, and when multiplied with the linear features of the state and summed, produce the final value function.

More formally, a tabular linear function TLF is represented by Equation 5, which is a sum of n terms. Each term is a product of a linear feature ϕ_i and a weight θ_i. The linear features ϕ_i need not be distinct, although they usually are. Each weight θ_i is a function of m_i nominal features $f_{i,1}, \ldots, f_{i,m_i}$.

$$h(s) = \sum_{i=1}^{n} \theta_i(f_{i,1}(s), \ldots, f_{i,m_i}(s))\phi_i(s) \tag{5}$$

A TLF reduces to a linear function when there are no nominal features, i.e., $\theta_1, \ldots, \theta_n$ are scalar values. One can also view any TLF as a purely linear function where there is a term for every possible set of values of the nominal features:

$$h(s) = \sum_{i=1}^{n} \sum_{k \in K} \theta_{i,k}\phi_i(s) I(f_i(s) = k) \tag{6}$$

Here $I(f_i(s) = k)$ is 1 if $f_i(s) = k$ and 0 otherwise. $f_i(s)$ is an abbreviation of the m_i-component function in Equation 5, K is the set of all of its possible values. TLFs reduce to a table when there is a single term and no linear features, i.e., $n = 1$ and $\phi_1 = 1$ for all states. They reduce to tile coding or coarse coding when there are no linear features, but there are multiple terms, i.e., $\phi_i = 1$ for all i and $n \geq 1$. The nominal features of each term can be viewed as defining a tiling or partition of the state space into non-overlapping regions and the terms are simply added up to yield the final value of the state [1].

Consider, for example, one particular application of TLFs to our product delivery domain. We can represent the value function $h(s)$ by Equation 7:

$$h(s) = \sum_{t=1}^{k} \sum_{x=1}^{n} \theta_{t,x}(p_t, l_t, i_x) \tag{7}$$

where there are k trucks, n shops, and no linear features. The value function has kn terms, each term corresponding to a truck-shop pair (t, x). The nominal features are truck position p_t, truck load l_t, and shop inventory i_x. This is the form of TLF we use in our experiments.

In general, the value function $h(.)$ in ARL is represented as a parameterized functional form of Equation 5 with weights $\theta_1, \ldots, \theta_n$ and linear features $\phi_1, \ldots, \phi_n$. Each weight θ_i is a function of m_i nominal features $f_{i,1}, \ldots, f_{i,m_i}$.

Then each θ_i is updated using the following equation:

$$\theta_i(f_{i,1}(s), \ldots, f_{i,m_i}(s)) \leftarrow \theta_i(f_{i,1}(s), \ldots, f_{i,m_i}(s)) + \beta(TDE(s))\nabla_{\theta_i} h(s) \tag{8}$$

where $\nabla_{\theta_i} h(s) = \phi_i(s)$ and β is the learning rate.

The above update suggests that the value of $h(s)$ would be adjusted to reduce the temporal difference error in state s. The update is exactly the same as that of linear value functions except that only those parameters that belong to the entry that corresponds to the current state's nominal features get the update in proportion to the value of the linear feature.

4.2 Hill Climbing for Action Space Search

To mitigate the exponential growth of the joint action space and the time required to search this action space, we implemented a simple form of hill climbing which greatly sped up the process without a loss in the quality of the resulting policy. We used hill climbing only during training, and used complete action-space search during the evaluation.

Note that every joint action $\boldsymbol{a}$ is a vector of sub-actions, each by a single truck, i.e., $\boldsymbol{a} = (a_1, \ldots, a_k)$. This vector is initialized with all "wait" actions. Starting at a_1, we consider a neighborhood of 8 actions (one for each possible action a_1 may take, other than the action it is currently set to), and $\boldsymbol{a}$ is set to the best action. This process is repeated for each truck $a_2, \ldots, a_k$. The process then starts over at a_1, repeating until $\boldsymbol{a}$ has converged to a local optimum (all agent actions stay the same on one pass over the actions).

4.3 ASH-Learning

One of the drawbacks of model-based RL methods is that they require stepping through all possible next states of a given action to compute the expected value of the next state. Optimizing this step improves the speed of the algorithm considerably. Consider the fact that we need to compute the term $\sum_{s'=1}^{N} p(s'|s,u)h(s')$ in Equation 3 to compute the Bellman error and update the parameters. Since there are usually an exponential number of possible next states in parameters such as the number of shops, doing this calculation by brute-force is expensive.

A method for optimizing the calculation of the expectation is an algorithm we call *ASH-Learning*, or Afterstate H-Learning. This is based on the notion of *afterstates* [1] also called "post-decision states" [2]. We create afterstates by conceptually splitting the effects of an agent's action into "action-dependent" and "action-independent" (or environmental) effects. The afterstate is the state that results by taking into account only the action-dependent effects. We can view the progression of states/afterstates as $s \xrightarrow{a} s_a \rightarrow s' \xrightarrow{a'} s'_{a'} \rightarrow s''$. The "$a$" suffix used here indicates that s_a is the afterstate of state s and action a. The action-independent effects of the environment have created state s' from afterstate s_a. The agent chooses action a' leading to afterstate $s'_{a'}$ and receiving reward $r(s', a')$. The environment again stochastically selects a state, and so on. The h-values may now be redefined in these terms:

$$h(s_a) = E(h(s')) \tag{9}$$

$$h(s') = \max_{u \in U(s')} \left\{ r(s', u) + \sum_{s'_u = 1}^{N} p(s'_u | s', u) h(s'_u) \right\} - \rho \tag{10}$$

If we substitute Equation 10 into Equation 9, we obtain this Bellman equation:

$$h(s_a) = E\left[\max_{u \in U(s')} \left\{ r(s', u) + \sum_{s'_u = 1}^{N} p(s'_u | s', u) h(s'_u) \right\} - \rho \right] \tag{11}$$

```
1. Find an action $u \in U(s')$ that maximizes $\left\{ r(s',u) + \sum_{s'_u=1}^{N} p(s'_u|s',u)h(s'_u) \right\}$
2. Take an exploratory action or a greedy action in the state $s'$. Let $a'$ be the action
   taken, $s'_{a'}$ be the afterstate, and $s''$ be the resulting state.
3. Update the model parameters $p(s'_{a'}|s',a')$ and $r(s',a')$ using the immediate re-
   ward received.
4. If a greedy action was taken, then
   (a) $\rho \leftarrow (1-\alpha)\rho + \alpha(r(s',a') - h(s_a) + h(s'_{a'}))$
   (b) $\alpha \leftarrow \frac{\alpha}{\alpha+1}$
5. $h(s_a) \leftarrow (1-\beta)h(s_a) + \beta \left( \max_{u \in U(s')} \left\{ r(s',u) + \sum_{s'_u=1}^{N} p(s'_u|s',u)h(s'_u) \right\} - \rho \right)$
6. $s' \leftarrow s''$
7. $s_a \leftarrow s'_{a'}$
```

Fig. 2. The ASH-learning algorithm. The agent executes steps 1-7 when in state s'.

The s'_u notation indicates the afterstate obtained by taking action u in state s'. We estimate the expectation of equation 11 via sampling in step 5 of Figure 2. Since this avoids looping through all possible next states, the algorithm is faster. In our domain, the afterstate is deterministic given the agent's actions, but the stochastic effects of customer actions are unknown. Hence we do not need to learn $p(s'_u|s',u)$, providing a significant savings in computation time.

The temporal difference error for the ASH-learning algorithm is given by:

$$TDE(s_a) = \max_{u \in U(s')} \left\{ r(s',u) + \sum_{s'_u=1}^{N} p(s'_u|s',u)h(s'_u) \right\} - \rho - h(s_a) \qquad (12)$$

which we use in Equation 8 when using TLFs for function approximation.

5 Experimental Results

We conducted several experiments testing the methods discussed in Section 4. Tests are averaged over 30 runs of 10^6 time steps for all results. Runs are divided into 20 phases of 48,000 training steps and 2,000 evaluation steps each. During evaluation steps, exploration is turned off and complete search is used to select actions. 4 trucks and 5 shops were used in all the tests. In Figure 3 we compare the results of H-learning and ASH-learning, and complete search of the joint action space vs. hill climbing search. We also compare these results to a fairly sophisticated handcoded non-learning greedy algorithm. This algorithm works by first prioritizing the shops by the expected time until each shop is empty due to customer actions, and then assigning trucks to the highest-priority shops. Once an assignment is made it is straightforward to assign each truck an optimal action. Our results show that ASH-learning outperforms the hand-coded algorithm and H-learning, converging faster to a better average reward.

From Table 1, we see that ASH-learning is very successful at ameliorating the explosion in stochastic branching factor. The largest gains in execution time

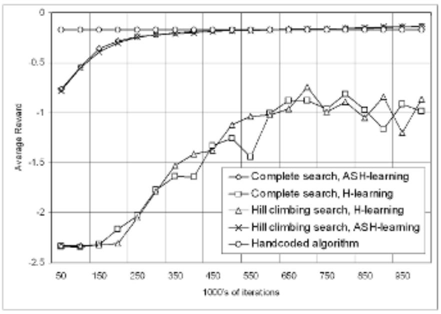

Fig. 3. Comparison of Handcoded algorithm, complete search, Hill climbing, and H- and ASH-learning

Table 1. Comparison of execution times for one run

Search	Algorithm	Seconds
Complete	H-learning	148
Complete	ASH-learning	92
Hill climbing	H-learning	26
Hill climbing	ASH-learning	15

were seen using hill climbing. Combined with ASH-learning, this led to speedups of an order of magnitude. This is because the average number of joint actions considered using hill climbing was about 44, whereas the number of legal truck actions considered by complete search was about 385. Despite this, there was no significant difference in the rate of convergence or the value of the final learned policy. As the number of agents increases beyond 4, we would expect to see even greater improvements in execution times of hill climbing search.

We obtained similar good results using Tabular Linear Functions and ASH-learning in other domains we tested including a taxi domain which is popular in hierarchical RL [10] and a competitive team game domain (which requires strong coordination between agents). We have also developed a multi-agent version of ASH-learning which allowed us to scale up to 10 agents in the team game domain without significant slow down.

6 Discussion and Future Work

We illustrated the three curses of dimensionality of reinforcement learning and showed effective techniques to address them in certain domains. TLFs offer an attractive alternative to other nonlinear forms of function approximation such as neural nets. They converge faster and allow meaningful prior knowledge to be provided. Hill climbing is a cheap but effective technique to mitigate the action-space explosion due to multiple agents. Another promising avenue to explore is the coordination graph approach, where the value function is decomposed in a way that makes the coordination opportunities among agents explicit [11].

We introduced ASH-learning, which is an afterstate version of model-based real-time dynamic programming. It is similar to R-learning in that action- independent effects are not learned or used [12]. However, the value function is state-based, so is more compact than R-learning, much more so for multiple agents. Thus it combines the nice features of both model-based and model-free methods and has proved itself quite well in our domain. Any afterstate-based method is expected to be quite effective in domains where action-dependent effects can be conceptually separated from environmental effects.

In summary, our methods and results add to the accumulating evidence of the effectiveness of average-reward RL, and suggest that the explosions in state space, action space, and high stochasticity may all be ameliorated.

Acknowledgments

This work was supported by the National Science Foundation under grant number: IIS-0329278. The authors would like to thank Hong Tang for his initial experiments and code, and Alan Fern, Neville Mehta, Jason Tracy, and Rasaratnam Logendran for many useful discussions.

References

1. Sutton, R.S., Barto, A.G.: Reinforcement learning: an introduction. MIT Press (1998)
2. Powell, W.B., Van Roy, B.: Approximate Dynamic Programming for High-Dimensional Dynamic Resource Allocation Problems. In Si, J., Barto, A.G., Powell, W.B., Wunsch, D., eds.: Handbook of Learning and Approximate Dynamic Programming. Wiley-IEEE Press, Hoboken, NJ (2004)
3. Van Roy, B., Bertsekas, D.P., Lee, Y., Tsitsiklis, J.N.: A Neuro-Dynamic Programming Approach to Retailer Inventory Management. In: Proceedings of the IEEE Conference on Decision and Control. (1997)
4. Secamondi, N.: Comparing Neuro-Dynamic Programming Algorithms for the Vehicle Routing Problem with Stochastic Demands. Computers and Operations Research **27**(11-12) (2000)
5. Secamondi, N.: A Rollout Policy for the Vehicle Routing Problem with Stochastic Demands. Operations Research **49**(5) (2001) 768–802
6. Strens, M., Windelinckx, N.: Combining planning with reinforcement learning for multi-robot task allocation. In: Lecture Notes in Computer Science. Volume 3394. (2005) 260–274
7. Puterman, M.L.: Markov Decision Processes: Discrete Dynamic Stochastic Programming. John Wiley (1994)
8. Tadepalli, P., Ok, D.: Model-based Average Reward Reinforcement Learning. Artificial Intelligence **100** (1998) 177–224
9. Bräysy, O., Gendreau, M.: Vehicle Routing Problem with Time Windows, Part II: Metaheuristics. Working Paper, SINTEF Applied Mathematics, Department of Optimisation, Norway (2003)
10. Ghavamzadeh, M., Mahadevan, S.: Learning to communicate and act using hierarchical reinforcement learning. In: AAMAS, IEEE Computer Society (2004) 1114–1121
11. Guestrin, C., Lagoudakis, M., Parr, R.: Coordinated reinforcement learning. In: Proceedings of the 19th International Conference on Machine Learning, San Francisco, CA, Morgan Kaufmann (2002)
12. Schwartz, A.: A Reinforcement Learning Method for Maximizing Undiscounted Rewards. In: Proceedings of the 10th International Conference on Machine Learning, Amherst, Massachusetts, Morgan Kaufmann (1993) 298–305