

Scaling Average-reward Reinforcement Learning for Product Delivery

Scott Proper and Prasad Tadepalli

School of Electrical Engineering and Computer Science

Oregon State University

Corvallis, OR 97331 3202

proper@cs.orst.edu

tadepall@cs.orst.edu

Abstract

Reinforcement learning in real-world domains suffers from three curses of dimensionality: explosions in state space and action space, and high stochasticity. We give partial solutions to each of these curses that provide order-of-magnitude speedups in execution time over standard approaches. We demonstrate our methods in the domain of product delivery. We present experimental results on refinements of a model-based average-reward algorithm called H-Learning.

1. Introduction

Reinforcement Learning (RL) provides a nice framework to model a variety of real-world stochastic optimization problems (Sutton & Barto 1998). However, textbook approaches to real-world RL suffer from what we call the three “curses of dimensionality”: explosions in state space and action space, and the high branching factor due to stochasticity (Powell & Van Roy 2004). In this paper, we use the RL framework to solve these three curses in the context of real-time delivery of products using multiple vehicles with stochastic demands. While RL has been applied separately to inventory control (Van Roy *et al.* 1997) and vehicle routing (Secamondi 2000; 2001; Powell & Van Roy 2004) 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.

The delivery problem we investigate is as follows: there is one distribution center and several shops. Products are delivered from the distribution center to the shops by several trucks. We assume that there is a single type of product. The current status of the inventory levels of the shops, and the loads and locations of the trucks at any time are known to the scheduling system. The task of the system is to decide at each time-step the next action for each truck, including going to a different location, unloading, and waiting. Since the system is only given the levels of the inventories and the current load levels and locations of the trucks, and is required to determine the best actions for all the trucks, it addresses the problems of inventory control, vehicle assignment, and routing simultaneously. While the actions of the

trucks themselves only have deterministic effects, they happen concurrently with customer actions of consuming the products from the shops, which are highly stochastic. As a result, our problem is a stochastic Markov Decision Problem (MDP) with two kinds of costs: 1) the cost of moving the truck from one location to the other, which combines the fuel, vehicle, and the driver costs; and 2) the *stockout cost*, which is the lost profit due to not satisfying the customer when a customer finds the shop empty.

While it is natural to minimize the expected total cost in stochastic domains where the planning horizon is finite, in most recurrent domains including real-time delivery, the natural optimization criterion is to maximize the expected *average reward per time step*. In this paper we investigate the effectiveness of a model-based Average-reward Reinforcement Learning (ARL) method called H-Learning, and its variants (Tadepalli & Ok 1998). H-Learning works by learning explicit models of the actions and the environment as well as a value function over the states. The models represent how the state of the world changes with the actions chosen by the system. In our case, since the direct effects of actions by the trucks are known already, the distribution of consumption at each shop is learned. The value function represents the *bias* of each state, defined as the best expected reward when starting from that state over and above what would be expected on the average over the infinite horizon. Given the models and the optimal value function over all states, a one-step look-ahead search over all available actions of all trucks would yield the optimal decision.

Unfortunately, H-Learning, or any RL method that depends on storing the optimal value function and action models as tables, does not scale to large state-spaces. Three computational obstacles prevent the use of standard approaches when dealing with problems with many variables. First, the state space (and time required for convergence) grows exponentially in the number of variables, e.g. the number of shops and trucks in our domain. This makes computing the value function impractical or impossible in terms of both memory and time. Second, the space of possible actions is exponential in the number of agents, i.e. trucks, so even one-step look-ahead search is computationally expensive. Lastly, exact computation of the expected value of the next state is slow, as the number of possible future states is exponential in the number of variables, i.e., the number of shops. These

1. $GreedyActions(i) \leftarrow$ All actions $u \in U(i)$ that maximize
 - (a) $\{r_\delta(i, u) + \theta_{0,v} + \sum_{l=1}^n \theta_{l,v} E(\phi_{l,i} | i, u)\}$ (If using linear function approximation)
 - (b) $\{r_\delta(i, u) + \sum_{j=1}^N p_\delta(j|i, u)h(j)\}$ (otherwise)
2. Take an exploratory action or a greedy action in the current state i . Let a be the action taken, k be the resulting state, and r_{imm} be the immediate reward received.
3. Update the model parameters δ for $p_\delta(j|i, a)$ and $r_\delta(i, a)$
4. If $a \in GreedyActions(i)$, then
 - (a) $\rho \leftarrow (1 - \alpha)\rho + \alpha(r_\delta(i, a) - h(i) + h(k))$
 - (b) $\alpha \leftarrow \frac{\alpha}{\alpha+1}$
- 5.(a) $\vec{\theta}_v = \vec{\theta}_v + \beta (\max_{u \in U(i)} \{r_\delta(i, u) + \theta_{0,v} + \sum_{l=1}^n \theta_{l,v} E(\phi_{l,i} | i, u) - \rho\} - h(i)) \vec{\phi}_i$
(If using linear function approximation)
- (b) $h(i) \leftarrow \max_{u \in U(i)} \{r_\delta(i, u) + \sum_{j=1}^N p_\delta(j|i, u)h(j)\} - \rho$ (otherwise)
6. $i \leftarrow k$

Figure 1: The H-learning algorithm. The agent executes steps 1-6 when in state i . (Tadepalli & Ok 1998)

three obstacles are referred to as the three ‘‘curses of dimensionality’’ (Powell & Van Roy 2004).

We introduce methods to deal with each of these three curses of dimensionality. To solve the exploding state-space problem, we use a function approximator which stores the value function as a set of distinctly scoped linear functions, which achieves a high degree of compression and faster learning. We compare our linear function approximator results to table-based methods and similarly distinctly scoped neural networks. To reduce the computational cost of searching the action space, which is exponential in the number of agents, we use a stochastic hill climbing algorithm that runs significantly faster than exhaustive search and scales to a larger number of agents without sacrificing the quality of the solutions. To eliminate the exponential cost of computing the expected value of the next state due to multiple resulting states of a stochastic action, we take advantage of linear function approximation. We also introduce an alternative method that is based on the idea of ‘‘afterstates’’ (Sutton & Barto 1998), which separates the deterministic effects of an action from its stochastic effects, and may be used with any type of function approximation.

In the next section, we discuss Average-reward Reinforcement Learning (ARL) and H-learning. In Section 3 we describe our method of function approximation and our solutions to the three curses of dimensionality. In Section 4, we present experimental results and in Section 5 we describe possibilities for future work.

2. Average-Reward Reinforcement Learning

We assume that the learner’s environment is modeled by a Markov Decision Process (MDP). An MDP is described by a discrete set S of N states, and a discrete set of actions, A . The set of actions that are applicable in a state i are denoted by $U(i)$ and are called *admissible*. The Markovian assumption means that an action u in a given state $i \in S$ results in state j with some fixed probability $p_\delta(j|i, u)$. There is a finite immediate reward $r_\delta(i, j, u)$ for executing an ac-

tion u in state i resulting in state j . Here δ denotes a set of parameters that fully determine the state-transition function and the immediate reward function. Time is treated as a sequence of discrete steps. A *policy* μ is a mapping from states to actions, such that $\mu(i) \in U(i)$. We only consider policies that do not change over time, which are called ‘‘stationary policies’’ (Puterman 1994). In Average-reward Reinforcement Learning (ARL), we seek to optimize the average expected reward per step over time t as $t \rightarrow \infty$, which is called the *gain*. 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:

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

The goal of ARL is to control the MDP in a way that achieves near-optimal gain by executing actions, receiving rewards and learning from them. Ideally, we would like to find a policy that optimizes the gain, i.e., a *gain-optimal policy*. The expected total reward in time t for optimal policies depends on the starting state s and can be written as the sum $\rho(s) \cdot t + h_t(s)$, where $\rho(s)$ is its gain. The Cesaro’s limit of the second term $h_t(s)$ as $t \rightarrow \infty$ is called the *bias* of state s and is denoted by $h(s)$. In communicating MDPs, where every state is reachable from every other state, the optimal gain ρ^* is independent of the starting state (Puterman 1994). ρ^* and the biases of the states under the optimal policy satisfy the following Bellman equation:

$$h(i) = \max_{u \in U(i)} \left\{ r_\delta(i, u) + \sum_{j=1}^N p_\delta(j|i, u)h(j) \right\} - \rho^* \quad (2)$$

The optimal policy chooses actions that maximize the right hand side of the above equation. The classical dynamic programming (DP) methods to solve the above equation, including *value iteration* and *policy iteration*, depend on knowing the probability transition models $p_\delta(j|i, u)$ and the

immediate rewards $r_\delta(i, u)$. Reinforcement Learning (RL) methods need to learn these models on-line by executing actions, observing the next states, and receiving rewards (Sutton & Barto 1998). We use an ARL method called ‘‘H-Learning’’ which is ‘‘model-based’’ in that it uses explicitly represented action models $p_\delta(j|i, u)$ and $r_\delta(i, u)$. They may be learned by a straightforward Monte Carlo sampling using the following updates:

$$p_\delta(k|i, a) \leftarrow \frac{N(\text{state}_t = i, \text{action}_t = a, \text{state}_{t+1} = k)}{N(\text{state}_t = i, \text{action}_t = a)} \quad (3)$$

$$r_\delta(i, a) \leftarrow r_\delta(i, a) + \frac{(r_{imm} - r_\delta(i, a))}{N(\text{state}_t = i, \text{action}_t = a)} \quad (4)$$

Where the $N()$ notation represents the number of times the conditions in the parentheses have held. For example, $N(\text{state}_t = i, \text{action}_t = a)$ is the number of times the H-learning algorithm has been in state i and taken action a .

A popular alternative method involves the use of factored representations in the form of dynamic Bayesian networks (Russell & Norvig 1995) to calculate the next-state probabilities and rewards. To see this, consider equation (3) above. Here, the terms k and i are states. However many of the state variables may be independent of each other. The simple dynamic Bayesian network in Figure 2 illustrates this. Here, we can see that shop inventory levels, truck load levels, and truck locations are independent of all other variables except their own values in the previous time step, and the action. The values of truck locations and load levels are deterministic, so only one example is required to learn these. Shop inventory levels are stochastic, and so we may learn distributions for these by adapting equation (3), where $s_{t,l}$ represents the inventory level of shop l at time t :

$$\delta_{l,x,y,a} = p(s_{t+1,l} = y | s_{t,l} = x, \text{action}_t = a) = \frac{N(s_{t,l} = x, \text{action}_t = a, s_{t+1,l} = y)}{N(s_{t,l} = x, \text{action}_t = a)} \quad (5)$$

Here we calculate the probability that the inventory level of shop $l = y$ in the next time step, given that its current inventory level is x and that we took action a . In our experiments, the model parameters δ are given. It would be easy to learn these parameters via Equation (5) if they are not known. δ can be used to estimate $p_\delta(k|i, a)$ using standard Bayesian network inference algorithms.

The H-learning algorithm then employs the ‘‘certainty equivalence principle’’ by using the current estimates as the true values while updating the h -value of the current state i according to the equation (step 5.b. of Figure 1):

$$h(i) \leftarrow \max_{u \in U(i)} \left\{ r_\delta(i, u) + \sum_{j=1}^N p_\delta(j|i, u) h(j) \right\} - \rho \quad (6)$$

Unlike the classical DP methods that update all states in a sweep, at any time the RL algorithms such as ours only update the value of the current state i , and not the values of other states. To make sure that all state-values are updated

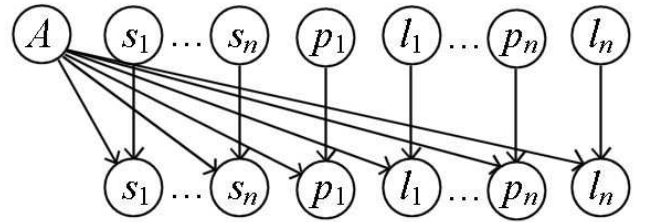


Figure 2: Dynamic Bayesian network showing the influences of action A , shop levels $s_1 \dots s_n$, truck loads $l_1 \dots l_n$, and truck positions $p_1 \dots p_n$ at time t on the variables at time $t + 1$.

until they converge, Reinforcement Learning methods require ‘‘exploration actions’’ that ensure that every action in every state is executed with some non-zero probability (step 2 of Figure 1). However, to get a good online reward, it should not deviate too much from the greedy policy as computed in step 1 of the algorithm in Figure 1. One issue that still needs to be addressed in Average-reward RL is the estimation of ρ^* , the optimal gain. Since the optimal gain is unknown, we use ρ , an estimate of the average reward of the current greedy policy, instead. From Equation (2), it can be seen that $r_\delta(i, a) + h(k) - h(i)$ gives an unbiased estimate of ρ^* , when action a is greedy. This is what is estimated in Step 4 of Figure 1.

We note here that the H-learning algorithm shown in Figure 1 is optimized compared to the original algorithm in (Tadepalli & Ok 1998): the calculation of the greedy actions has been moved to step 1. Since the model parameters are known, we do not need to redo the search in step 5. If the models are not given but are learned, it may become necessary to use two searches; however because the model parameters eventually converge, this should not matter in the long run.

3. Three Curses of Dimensionality

Consider the supplier of a product such as bread that needs to be supplied to several shops from a warehouse using several delivery trucks. What is an efficient policy for ensuring the stores remain supplied? Trucks must not move unnecessarily, but must still make deliveries on time to prevent stockouts. We developed a simple model and experimented with an instance of the problem shown in Figure 3. We considered 5 shops, one centrally located depot, and 4 other intermediate locations where trucks can change their routes. We assumed it takes one unit of time to go from any location to its adjacent location or to execute an unload action. The shop inventory levels and truck load levels are discretized into 5 levels 0-4. With 2 trucks and 10 locations, this gives us a state space size of $(5^5)(5^2)(10^2) = 7,812,500$, and with 5 trucks and 10 locations, it gives us a size of $(5^5)(5^5)(10^5) = 976,592,500,000$. State space size grows exponentially in the number of trucks and the number of shops. Each truck has 9 actions available at each time step: unload 25%, 50%, 75%, or 100% of its load, move in one of the four available directions, or do nothing. This is a total

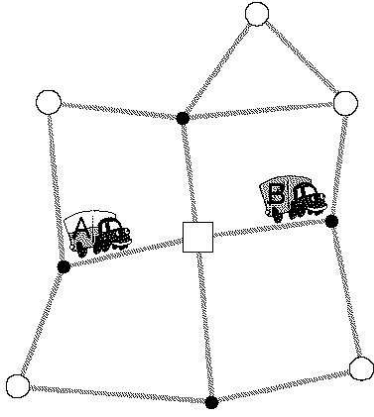


Figure 3: The Product Delivery Domain: The square in the center is the depot and the large circles are the shops.

of $9^2 = 81$ actions for 2 trucks, or as many as $9^5 = 59,049$ actions for 5 trucks. Trucks are loaded automatically upon reaching the depot. A small negative reward of -0.1 is given for every “move” action a truck may take to reflect the fuel, maintenance, and the driver costs. There is a small probability of a visit by a customer at every time step, which varies from shop to shop. If a customer enters a store and finds the shelves empty, the system receives a penalty of -20 . Because there are 5 shops, each of which may or may not be visited by a customer each time step, this gives us up to $2^5 = 32$ possible next states each time step. We call this the stochastic branching factor – the maximum number of possible next states for any state-action pair.

3.1 Explosion of the State Space

Table-based methods such as H-Learning do not scale to such large state spaces as the delivery domain both due to limitations of space and convergence speed. The value function needs to be represented more compactly using function approximation to make it scale to large domains. We have experimented with two methods of function approximation: neural networks and linear functions.

Linear function approximators are the simplest and probably the fastest. Unfortunately, however, in many cases including ours, the optimal value function is highly nonlinear and cannot be captured by a single global linear value function in the state features. Instead, we used a function approximation scheme based on multiple linear functions. In particular, we break down the value function into several linear functions, one for each set of values of the nominal attributes of the state. Each such piece of the value function is linear in the remaining features.

For example, in the delivery domain, it can be argued that the optimal value function is approximately monotonically non-decreasing in certain variables of the state such as the inventory levels of the shops and the trucks, for any given set of locations of the trucks. This is because the chance of a stockout is smaller when the inventories are higher than when they are lower. Similarly one has more options when

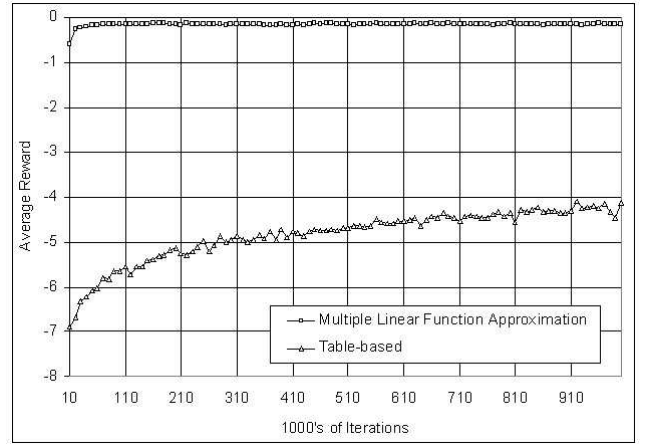


Figure 4: Multiple linear function approximation vs. table-based approach

the trucks have more load than when they have less load. We made a further simplifying assumption that the value function is *linear* in all these primitive features of the state, i.e., s_1, \dots, s_5 , that represent the shop inventory levels, and l_1, \dots, l_5 , that represent the truck loads. A third assumption is that the value function is separable in that it does not have cross-terms such as $l_1 l_2$. Since the locations of the trucks are nominal attributes, we assume a different linear function for each possible 5-tuple of positions p_1, \dots, p_5 for the 5 trucks. Thus the entire value function is represented as a set of linear functions, with each function covering one set of locations of trucks. Hence m^k linear functions are needed to represent the h -function, where k is the number of trucks and m is the total number of locations. Our test domain has 10 ordered attributes – 5 shop inventory levels and 5 truck load levels – so each linear function has 11 weights (adding one weight, θ_0 , that represents the constant term). There are 10^5 different linear functions, thus reducing the total number of parameters to be learned to 1.1 million, nearly a million-fold reduction from the table-based approach. As we will later show, this number can be further reduced by ignoring truck identity.

The h -value is represented as a parameterized functional form with parameter vectors $\vec{\theta}_v$, one for each possible set v of values of the nominal attributes. In our domain, v is actually a vector that represents the locations of the trucks. Corresponding to every state i , there is a vector of n ordered features, $\vec{\phi}_i = (1, \phi_{1,i}, \phi_{2,i}, \dots, \phi_{n,i})^T$. For each set of values of v , the parameter vectors, $\vec{\theta}_v = (\theta_{0,v}, \theta_{1,v}, \theta_{2,v}, \dots, \theta_{n,v})^T$, have the same number of components as $\vec{\phi}_i$. The approximate h -value function is given by:

$$h(i) = \vec{\theta}_v \cdot \vec{\phi}_i = \sum_{j=0}^n \theta_{j,v} \phi_{j,i} \quad (7)$$

where v is the set of values of the nominal features in state

i . Then $\vec{\theta}_v$ is updated using the following equation:

$$\vec{\theta}_v \leftarrow \vec{\theta}_v + \beta \left(\max_{u \in U(i)} \left\{ r_\delta(i, u) + \sum_{j=1}^N p_\delta(j|i, u) h(j) - \rho \right\} - h(i) \right) \nabla_{\vec{\theta}_v} h(i) \quad (8)$$

where $\nabla_{\vec{\theta}_v} h(i) = \nabla_{\vec{\theta}_v} (\vec{\theta}_v \cdot \vec{\phi}_i) = \vec{\phi}_i$ and β is the learning rate. The above update suggests that the value of $h(i)$ would be adjusted toward the result of one-step backed up value of the next state.

A second domain-specific optimization allows us to reduce the number of linear functions or neural networks necessary to represent the value function. The figure of m^k functions stated in the previous section assumes that a specific identity is associated with each truck. This is more information than is actually necessary: the identification of the trucks should not matter to the value function — only their load levels should. For example, consider Figure 3, where truck A is at location p_1 and truck B is at location p_2 . The value of this state is identical to the value of the state where the labels of the trucks (but not their loads) have been swapped: truck B is at location p_1 with 50% load and truck A is at location p_2 with full load. If we merge these states we can reduce the space requirement for representing the h -function, which in turn leads to faster convergence. Instead of m^k linear functions, we now require $\binom{m-1+k}{k}$.

For 2 trucks, which previously required 100 linear functions, we now require only 55. For 5 trucks, previously requiring 10^5 functions, we only need 2002 linear functions. This reduces the number of learnable weights to 22,022 from 1.1 million in the 5-truck case.

Figure 4 compares the performance of the agent using our multiple linear function approximation vs. using simple table-based approaches. Two trucks and five shops were modeled. The results of 20 runs were averaged over 10^6 steps. Every 10,000 steps, 2000 steps were used to evaluate performance and calculate an average reward over those 2000 steps. During the evaluation time, only greedy actions were chosen. As can be seen from these results, the state space is so large, even for just 2 trucks, that table-based approaches converge too slowly to be practical. Using function approximation allows for much faster convergence.

3.2 Explosion of the Action Space

The second curse of dimensionality we encountered was the size of the action space. With 5 trucks, we have 9^5 joint actions. It is usually too expensive to consider each possible joint action at each step. Although we could do this in principle, it can take over a day of simulation time to finish learning using this method. We implemented a simple variation on hill climbing which significantly sped up the process. Like methods such as simulated annealing (Chiang & Russell 1996) and Tabu search (Glover 1989), this variation is what is called a metaheuristic method in the operations research literature: unlike standard local search techniques,

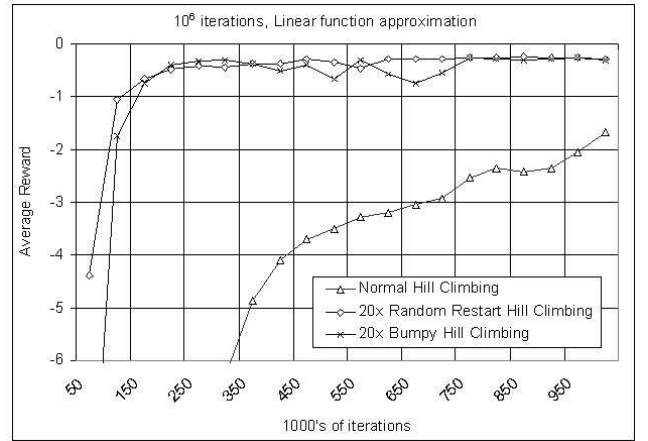


Figure 5: Comparison of three hill climbing approaches

metaheuristics allow deteriorating and even infeasible intermediate solutions during the search process in order to escape local optima (Bräysy & Gendreau 2003).

Note that every action a is a vector of individual “truck actions” $\vec{a} = (a_1, a_2, \dots, a_t)$. This vector is initialized with all “wait” actions. Starting at a_1 , each of the 8 other possible actions for that truck is considered, and \vec{a} is set to the best action. This gives us a neighborhood of 8 actions to consider at each step. Good results may also be achieved if a larger neighborhood of 40 actions (8 actions for each truck) are considered. This process is repeated for each truck $a_2 \dots a_n$. The process then starts over at a_1 , repeating until \vec{a} has converged to a local optimum. At this point, the solution is “bumped” out of its local optimum by randomly changing one of the actions. The search repeats, converging and being bumped several times. The hope is that the “bumping” action shakes the search out of local optima to states from which the global optimum can be reached. The overall best solution is always remembered and returned at the end. We chose this method because it is simple, fast, and performs well compared to normal hill climbing.

In Figure 5 we have plotted the results of the “bumpy” hill climbing approach vs. normal hill climbing starting from an all-wait action, and random restart hill climbing with 20 restarts. All tests were averaged over 10 runs and 10^6 steps. Every 50,000 steps, 2000 steps were used to evaluate performance and calculate an average reward over those 2000 steps. As in the previous experiments, during the evaluation only greedy actions were chosen. Normal hill climbing does not perform well at all. The bumpy hill climbing approach performs as well as random restart hill climbing, but the latter approach offers little to no benefit in computation time over exhaustive search. The bumpy hill climbing approach can be much faster than exhaustive search, while offering comparable results (see Figures 8 and 9). We gave random restart hill climbing 20 restarts, the first of which was always an all-wait action. For comparison, the normal hill climbing search considered an average of 50 actions in a search of the action space. The random restart hill climbing considered an

1. $GreedyActions(j) \leftarrow$ All actions $u \in U(j)$ that maximize $\{r_\delta(j, u) + h(j'_d)\}$
2. Take an exploratory action or a greedy action in the current state j . Let a be the action taken, j_d be the afterstate, k be the resulting state, and r_{imm} be the immediate reward received.
3. Update the model parameters $r_\delta(j, a)$
4. If $a \in GreedyActions(j)$, then
 - (a) $\rho \leftarrow (1 - \alpha)\rho + \alpha(r_\delta(j, a) - h(i_d) + h(j_d))$
 - (b) $\alpha \leftarrow \frac{\alpha}{\alpha+1}$
5. $h(i_d) \leftarrow (1 - \beta)h(i_d) + \beta \left(\max_{u \in U(i)} \{r_\delta(j, u) + h(j'_d)\} - \rho \right)$
6. $j \leftarrow k$
7. $i_d \leftarrow j_d$

Figure 6: The H_d -learning algorithm. The agent executes steps 1-7 when in state j .

average of 1170, and bumpy hill climbing considered 720. An exhaustive search of the action space that ignores the most obviously illegal of the 9^5 possible actions considered an average of 1345 actions. As the number of trucks increases, we would expect to see even greater improvement in execution times of hill climbing search over exhaustive search of the action space.

3.3 Stochastic Branching

One of the drawbacks of the 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. In domains like ours where the state is decomposed into several parts, one for each shop, step 1 of the H-learning algorithm (Figure 1) is very time-consuming. Anything that can be done to optimize this step improves the speed of the algorithm considerably. Consider the Bellman update from H-learning in step 5(b) of Figure 1, which is repeated below:

$$h(i) \leftarrow \max_{u \in U(i)} \left\{ r_\delta(i, u) + \sum_{j=1}^N p_\delta(j|i, u)h(j) \right\} - \rho \quad (9)$$

We see that to compute the term $\sum_{j=1}^N p_\delta(j|i, u)h(j)$, one needs to at least consider states whose number is exponential in the number of shops (in our case, with 5 stores and 2 possible customer actions, we need to consider $2^5 = 32$ next states). Note that the possible next states of an action at any state differ only in the shop inventory levels, since they are the only stochastic parts of the system.

Multiple linear function approximation allows one way to speed up evaluation of this step. Since the value of $h(j)$ is assumed to be linear in the shop inventory levels, we can write this as $\sum_{j=1}^N p_\delta(j|i, u)(\theta_{0,v} + \sum_{l=1}^n \theta_{l,v} \phi_{l,i})$ which can be rewritten as $\theta_{0,v} \cdot \sum_{j=1}^N p_\delta(j|i, u) + \sum_{l=1}^n \theta_{l,v} \sum_{j=1}^N p_\delta(j|i, u) \phi_{l,i}$ and be simplified to $\theta_{0,v} + \sum_{l=1}^n \theta_{l,v} E(\phi_{l,j}|i, u)$, where $E(\phi_{l,j}|i, u) = \sum_{j=1}^N p_\delta(j|i, u) \phi_{l,j}$ and represents the expected value of the ordered variable l in the next state under action u . It is directly estimated by on-line sampling. Instead of taking

time proportional to the number of possible next states, this would only take time proportional to the number of ordered variables, which is exponentially smaller. For example, if the current inventory level of shop l is 2, and the probability of inventory going down by 1 in this step is 0.2, and the probability of its going down by 2 or more is 0, then $E(\phi_{l,j}|i, u) = 2 - 1 * .2 = 1.8$. Thus, we obtain step 5.a of the H-learning algorithm by substituting $\theta_{0,v} + \sum_{l=1}^n \theta_{l,v} E(\phi_{l,j}|i, u)$ for $\sum_{j=1}^N p_\delta(j|i, u)h(j)$ in Step 5.b and combining it with (8) above.

A second method for optimizing the calculation of the expectation lies in the use of *afterstates* (Sutton & Barto 1998) also called “post-decision states” (Powell & Van Roy 2004). The afterstate is the state that occurs immediately following the action, and before the stochastic effects of the action occur. If we consider Figure 7, we can see that the progression in time of states/afterstates would be $i \rightarrow i_d \rightarrow j \rightarrow j_d \rightarrow k$. The “ d ” notation used here indicates that i_d is the afterstate of state i . The stochastic effects of the environment have created state j from afterstate i_d . The agent chooses action a_2 leading deterministically to afterstate j_d and receiving reward $r_\delta(j, a_2)$. The environment again stochastically selects state k . The h -values may now be redefined in these terms:

$$h(i_d) = E(h(j)) \quad (10)$$

$$h(j) = \max_{u \in U(j)} (r_\delta(j, u) + h(j'_d)) - \rho^* \quad (11)$$

If we substitute equation 11 into equation 10, we obtain the

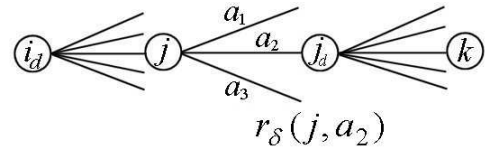


Figure 7: Progression of states (j and k) and afterstates (i_d and j_d).

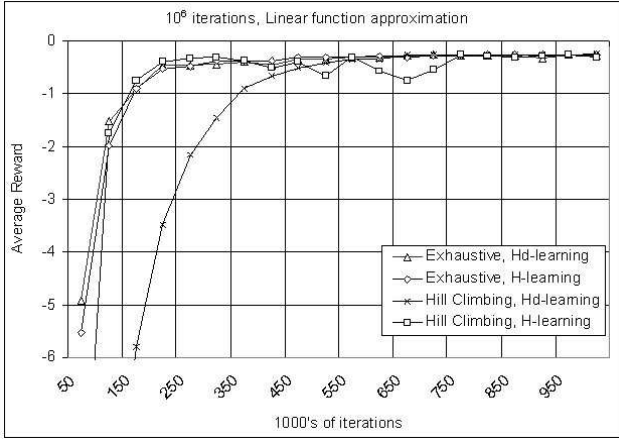


Figure 8: Comparison of Exhaustive search, Hill climbing, H- and H_d -learning for linear function approximation

following Bellman equation:

$$h(i_d) = E \left[\max_{u \in U(i)} \{r_\delta(j, u) + h(j'_d)\} - \rho^* \right] \quad (12)$$

Here the j'_d notation indicates the afterstate obtained by taking action u in state j . We estimate this expectation via sampling in step 5 of the H_d -learning algorithm. Since this avoids looping through all possible next states, it makes the algorithm much faster.

In our domain, the effects of the move and unload actions are deterministic. The afterstate is known with certainty, but the stochastic effects due to customer actions are unknown. Using afterstates to learn the expectation of the value of the next state takes advantage of our knowledge about the deterministic portion of the action. We have taken the “d” from “deterministic” and called this method H_d -learning.

A further advantage of this system is that because $p_\delta(j|i, u)$ is no longer used in calculations, it is no longer necessary to store this portion of the model. Because we must still learn (or have given to us) the reward model $r_\delta(i, u)$, H_d -learning is something of a compromise between model-free R-learning (Schwartz 1993) and model-based H-learning.

In step 5 of the H_d -learning algorithm, we learn the value of the afterstate i_d via sampling. If using linear function approximation, the update for step 5 would be:

$$\vec{\theta}_v = \vec{\theta}_v + \beta \left(\max_{u \in U(i)} \{r_\delta(j, u) + h(j'_d)\} - \rho - h(i_d) \right) \vec{\phi}_{i_d}$$

For our neural network, we used normal backpropagation with the temporal difference error as the error signal:

$$\vec{w} = \text{Backprop}(\vec{w}, \max_{u \in U(i)} \{r_\delta(j, u) + h(j'_d)\} - \rho - h(i_d))$$

4. Experimental Results

We conducted several experiments testing the methods discussed in the previous sections. Tests are averaged over 10

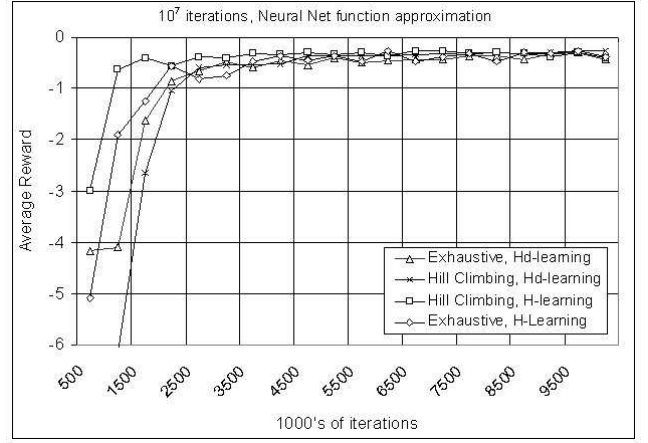


Figure 9: Comparison of Exhaustive search, Hill climbing, H- and H_d -learning for neural networks.

runs of 10^6 time steps each for the linear function approximator results, or 10^7 time steps for our neural network results. In all these tests 5 trucks and 5 shops were used. We do 8 separate tests, comparing linear and neural network function approximators, H-learning and H_d -learning, and exhaustive search of the action space vs. the bumpy hill climbing search. The hill climbing results were obtained by using bumpy hill climbing to learn on-line; then 2000 steps using exhaustive search were used to obtain the average reward for the policy learned so far. We were not successful using a single neural network to learn the entire value function. Instead we used 2002 networks, one for each possible combination of truck locations. Each network had 11 inputs scaled between 0 and 1, 5 truck load inputs, 5 shop inventory inputs, and a constant input always set to 1. Four hidden units were used in a single hidden layer, which was fully connected to the input and output units. A single output unit represented the h -value and was scaled between 0 and 1.

In all cases, linear function approximation performs much better than neural network function approximation. The number of iterations required for convergence was an order of magnitude smaller, and the time required to complete each iteration was also smaller.

H_d -learning performed as well as H-learning in most cases, although this method sometimes took more steps to converge. This is more than made up for the fact that with the neural network approach, order-of-magnitude speedups in computation time were observed. For the linear function approximation results, both methods of computing the expectation performed equally well.

Both Figures 8 and 9 also compare a bumpy hill climbing search of the action space vs. using the more typical exhaustive search of all possible actions during each iteration. Bumpy hill climbing performs as well as exhaustive search in these tests, while offering much improved speedups in execution time. This is an extremely encouraging result since it suggests that it may be possible to do less search and obtain just as good a result.

5. Summary and Future Work

We illustrated the three curses of dimensionality in applying reinforcement learning to real-world domains and showed ways to address them. The division of state features into nominal and ordered features and using them in different ways has been successful for us. Without this division, it may not have been practical to approximate the value function. We have demonstrated that the bumpy hill climbing technique can be useful in searching a large action space with multiple agents. Future research into multiple-agent reinforcement learning may benefit from a similar approach. Our first method for speeding up calculation of the expectation should be very useful in domains where linear function approximation is used. H_d -learning should be useful for domains in which a distinction can be drawn between the deterministic and stochastic effects of an action, allowing the use of afterstates.

There are several unresolved questions with the solutions to the three curses of dimensionality presented in this work. Continued research into improved function approximation methods could reduce the number of learned parameters necessary. Reformulation of the domain could reduce or eliminate the need for nominal attributes, and the need for hundreds of separate linear functions or neural networks. Continued research into hill climbing and similar approaches to searching the action space of multiple agents could be very useful, especially in domains with a large number of agents. It remains to be seen whether hill climbing search, and the bumpy hill climbing variant, will be successful in a wide range of domains.

It is fairly clear that our method of computing the expected h -value is correct for any domain where linear function approximation is used. It is not clear that H_d -learning will always be successful. What sorts of domains can this H -learning variant be applied to? We believe that the results of an action can be separated into deterministic and stochastic effects for many domains, but whether similar results may be achieved in these domains remains to be seen.

The domain itself is very similar to dynamic vehicle routing problems, but also includes shop inventories. Modern operations research methods can solve vehicle routing problems with hundreds of locations and many vehicles, but do not track shop inventories (Bräysy & Gendreau 2003). It remains to be seen if reinforcement learning will prove up to the task of scaling to such large state spaces while maintaining reasonable computation time. Adding more trucks, more locations, more shops, and more depots are all possible challenges. Even more difficult would be to add more product types: currently each truck carries only one type of item in its inventory. Scaling this up to two or more types creates a more difficult problem.

Instead of trucks moving between different locations in discrete “jumps”, allowing trucks to move with variable speed along highways interconnecting each intersection would be more realistic. This would require a generalization of our framework to semi-markov decision problems, and an event-based model of time, to allow trucks to arrive at shops and intersections at any time. Currently inventory and load levels are represented by 5 discrete values. Us-

ing real-valued shop inventory and truck load levels would be another interesting challenge, because it creates a real-valued action space.

In summary, we conclude that the explosions in state space, action space, and high stochasticity all have viable solutions. The experimental results in our domain give us confidence that we can make our approaches scale to larger and more practical problems.

6. Acknowledgements

This work was supported by the National Science Foundation under grant number: IIS-0098050. The authors would like to thank Hong Tang for his initial experiments and code, and Neville Mehta, Jason Tracy, and Rasaratnam Logendran for many useful discussions.

References

- Bräysy, O., and Gendreau, M. 2003. Vehicle Routing Problem with Time Windows, Part II: Metaheuristics. Working Paper, SINTEF Applied Mathematics, Department of Optimisation, Norway.
- Chiang, W. C., and Russell, R. A. 1996. Simulated Annealing Metaheuristics for the Vehicle Routing Problem with Time Windows. *Annals of Operations Research* 63:3–27.
- Glover, F. 1989. Tabu Search – Part I. *Journal on Computing* 1:190–206.
- Powell, W. B., and Van Roy, B. 2004. Approximate Dynamic Programming for High-Dimensional Dynamic Resource Allocation Problems. In Si, J.; Barto, A. G.; Powell, W. B.; and Wunsch, D., eds., *Handbook of Learning and Approximate Dynamic Programming*. Wiley-IEEE Press, Hoboken, NJ.
- Puterman, M. L. 1994. *Markov Decision Processes: Discrete Dynamic Stochastic Programming*. John Wiley.
- Russell, S., and Norvig, P. 1995. *Artificial Intelligence: A Modern Approach*. Prentice Hall.
- Schwartz, A. 1993. A Reinforcement Learning Method for Maximizing Undiscounted Rewards. In *Proceedings of the 10th International Conference on Machine Learning*, 298–305. Amherst, Massachusetts: Morgan Kaufmann.
- Secamondi, N. 2000. Comparing Neuro-Dynamic Programming Algorithms for the Vehicle Routing Problem with Stochastic Demands. *Computers and Operations Research* 27(11-12).
- Secamondi, N. 2001. A Rollout Policy for the Vehicle Routing Problem with Stochastic Demands. *Operations Research* 49(5):768–802.
- Sutton, R. S., and Barto, A. G. 1998. *Reinforcement learning: an introduction*. MIT Press.
- Tadepalli, P., and Ok, D. 1998. Model-based Average Reward Reinforcement Learning. *Artificial Intelligence* 100:177–224.
- Van Roy, B.; Bertsekas, D. P.; Lee, Y.; and Tsitsiklis, J. N. 1997. A Neuro-Dynamic Programming Approach to Retailer Inventory Management. In *Proceedings of the IEEE Conference on Decision and Control*.