ELSEVIER

Artificial Intelligence 100 (1998) 177–224

Artificial Intelligence

Model-based average reward reinforcement learning [1]

Prasad Tadepalli [a,*], DoKyeong Ok [b,2]

[a] *Department of Computer Science, Oregon State University, Corvallis, OR 97331, USA*

[b] *Korean Army Computer Center, NonSanSi DuMaMyeon BuNamRi, P.O.Box 29, ChungNam 320-919, South Korea*

Received 27 September 1996; revised 15 December 1997

Abstract

Reinforcement Learning (RL) is the study of programs that improve their performance by receiving rewards and punishments from the environment. Most RL methods optimize the discounted total reward received by an agent, while, in many domains, the natural criterion is to optimize the average reward per time step. In this paper, we introduce a model-based Average-reward Reinforcement Learning method called H-learning and show that it converges more quickly and robustly than its discounted counterpart in the domain of scheduling a simulated Automatic Guided Vehicle (AGV). We also introduce a version of H-learning that automatically explores the unexplored parts of the state space, while always choosing greedy actions with respect to the current value function. We show that this "Auto-exploratory H-Learning" performs better than the previously studied exploration strategies. To scale H-learning to larger state spaces, we extend it to learn action models and reward functions in the form of dynamic Bayesian networks, and approximate its value function using local linear regression. We show that both of these extensions are effective in significantly reducing the space requirement of H-learning and making it converge faster in some AGV scheduling tasks. © 1998 Published by Elsevier Science B.V.

Keywords: Machine learning; Reinforcement learning; Average reward; Model-based; Exploration; Bayesian networks; Linear regression; AGV scheduling

* Corresponding author. Email: tadepall@cs.orst.edu.

[1] Most of the work was done when both the authors were at Oregon State University.

[2] Email: okdo@unitel.co.kr.

PII S0004-3702(98)00002-2

1. Introduction

Reinforcement Learning (RL) is the study of programs that improve their performance at some task by receiving rewards and punishments from the environment. RL has been quite successful in automatic learning of good procedures for many tasks, including some real-world tasks such as job-shop scheduling and elevator scheduling [11,44,47]. Most approaches to reinforcement learning, including Q-learning [46] and Adaptive Real-Time Dynamic Programming (ARTDP) [3], optimize the total *discounted* reward the learner receives [18]. In other words, a reward that is received after one time step is considered equivalent to a fraction of the same reward received immediately. Discounted optimization criterion is motivated by domains in which reward can be interpreted as money that can earn interest in each time step. Another situation that it is well-suited to model is when there is a fixed probability that the run will be terminated at any given time for whatever reason. However, many domains in which we would like to use reinforcement learning do not have either the monetary aspect or the probability of immediate termination, at least in the time scales in which we are interested in. The natural criterion to optimize in such domains is the average reward received per time step. Even so, many people have used discounted reinforcement learning algorithms in such domains, while aiming to optimize the average reward [21,26]. One reason to do this is that the discounted total reward is finite even for an infinite sequence of actions and rewards. Hence, two such action sequences from a state can be compared by this criterion to choose the better one.

While mathematically convenient, in domains where these isn't a natural interpretation for the discount factor, discounting encourages the learner to sacrifice long-term benefits for short-term gains, since the impact of an action choice on long-term reward decreases exponentially with time. Hence, using discounted optimization when average-reward optimization is what is required could lead to suboptimal policies. Nevertheless, it can be argued that it is appropriate to optimize discounted total reward if that also nearly optimizes the average reward by using a discount factor which is sufficiently close to 1 [21,26]. This raises the question whether and when discounted RL methods are appropriate to use to optimize the average reward.

In this paper, we describe an Average-reward RL (ARL) method called H-learning, which is an undiscounted version of Adaptive Real-Time Dynamic Programming (ARTDP) [3]. We compare H-learning with its discounted counterpart, ARTDP, in the task of scheduling a simulated Automatic Guided Vehicle (AGV), a material handling robot used in manufacturing. Our results show that H-learning is competitive with ARTDP when the short-term (discounted with a small discount factor) optimal policy also optimizes the average reward. When short-term and long-term optimal policies are different, ARTDP either fails to converge to the optimal average-reward policy or converges too slowly if the discount factor is high.

Like ARTDP, and unlike Schwartz's R-learning [37] and Singh's ARL algorithms [38], H-learning is model-based, in that it learns and uses explicit action and reward models. We show that in the AGV scheduling domain H-learning converges in fewer steps than R-learning and is competitive with it in CPU time. This is consistent with the

previous results on the comparisons between model-based and model-free discounted RL [3,28].

Like most other RL methods, H-learning needs exploration to find an optimal policy. A number of exploration strategies have been studied in RL, including occasionally executing random actions, preferring to visit states that are least visited (counter-based) or executing actions that are least recently executed (recency-based) [45]. Other methods such as the Interval Estimation (IE) method of Kaelbling and the action-penalty representation of reward functions used by Koenig and Simmons incorporate the idea of "optimism under uncertainty" [17,20]. By initializing the value function of states with high values, and gradually decreasing them, these methods encourage the learner to explore automatically, while always executing greedy actions. We introduce a version of H-learning that uses optimism under uncertainty, and has the property of automatically exploring the unexplored parts of the state space while always taking a greedy action with respect to the current value function. We show that this "Auto-exploratory H-learning" converges more quickly to a better average reward when compared to H-learning under random, counter-based, recency-based, and Boltzmann exploration strategies.

ARL methods in which the value function is stored as a table require too much space and training time to scale to large state spaces. Model-based methods like H-learning also have the additional problem of having to explicitly store their action models and the reward functions, which is space-intensive. To scale ARL to such domains, it is essential to approximate its action models and value function in a more compact form.

Dynamic Bayesian networks have been successfully used in the past to represent the action models [12,35]. In many cases, it is possible to design these networks in such a way that a small number of parameters are sufficient to fully specify the domain models. We extended H-learning so that it takes the network structure as input and learns the conditional probabilities in the network. This not only reduces the space requirements for our method but also increases the speed of convergence in domains where learning the action models dominates the learning time.

Many reinforcement learning tasks, especially those with action models and reward functions that can be described by small Bayesian networks, have a uniform reward structure over large regions of the state space. In such domains, the optimal value function of H-learning is piecewise linear. To take advantage of this, we implemented a value function approximation method based on local linear regression. Local linear regression synergistically combines with learning of Bayesian network-based action models and improves the performance of H-learning in many AGV scheduling tasks. Combining Auto-exploratory H-learning with action model and value function approximation leads to even faster convergence in some domains, producing a very effective learning method.

The rest of the paper is organized as follows: Section 2 introduces Markov Decision Problems that form the basis for RL, and motivates Average-reward RL. Section 3 introduces H-learning and compares it with ARTDP and R-learning in an AGV scheduling task. Section 4 introduces Auto-exploratory H-learning, and compares it with some previously studied exploration schemes. Section 5 demonstrates the use of dynamic Bayesian networks and local linear regression to approximate the action models and the value function respectively. Section 6 is a discussion of related work and future research issues, and Section 7 is a summary.

2. Background

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 $P_{i,j}(u)$. There is a finite immediate reward $r_{i,j}(u)$ for executing an action u in state i resulting in state j. 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".

2.1. Discounted reinforcement learning

Suppose that an agent using a policy μ goes through states $s_0, \ldots, s_t$ in time 0 thru t, with some probability, accumulating a total reward $r^\mu(s_0, t) = \sum_{k=0}^{t-1} r_{s_k, s_{k+1}}(\mu(s_k))$. If the environment is stochastic, $r^\mu(s_0, t)$ is a random variable. Hence, one candidate to optimize is the expected value of this variable, i.e., the expected total reward received in time t starting from s_0, $E(r^\mu(s_0, t))$. Unfortunately, however, over an infinite time horizon, i.e., as t tends to ∞, this sum can be unbounded. The discounted RL methods make it finite by multiplying each successive reward by a discount factor γ, where $0 < \gamma < 1$. In other words, they optimize the expected discounted total reward given by:

$$f^\mu(s_0) = \lim_{t \to \infty} E\left(\sum_{k=0}^{t-1} \gamma^k r_{s_k, s_{k+1}}(\mu(s_k))\right). \tag{1}$$

It is known that there exists an *optimal discounted policy* μ^* that maximizes the above value function over all starting states s_0 and policies μ. It can be shown to satisfy the following recurrence relation [3,6]:

$$f^{\mu^*}(i) = \max_{u \in U(i)} \left\{ r_i(u) + \gamma \sum_{j \in S} p_{i,j}(u) f^{\mu^*}(j) \right\}. \tag{2}$$

Real-Time Dynamic Programming (RTDP) solves the above recurrence relation by updating the value of the current state i in each step by the right-hand side of the above equation. RTDP assumes that the action models $p_{*,*}(*)$ and the reward functions $r_*(*)$, are given. Adaptive Real-time Dynamic Programming (ARTDP) estimates the action model probabilities and reward functions through on-line experience, and uses these estimates as real values while updating the value function of the current state by the right-hand side of the recurrence relation above [3]. In dynamic programming literature, this method is called the certainty equivalence control [6]. It is a model-based method because it learns the action and reward models explicitly, and uses them to simultaneously learn the value function.

The model-free RL methods, such as Q-learning, combine the model learning and the value function learning into one step by learning a value function over state-action pairs.

The value of $Q(i,u)$ for state i and action u represents the discounted total reward of executing action u in state i and from then on following the optimal policy. Hence, the correct Q-values must satisfy the following relationship with $f^{\mu*}$.

$$Q(i,u) = r_i(u) + \gamma \sum_{j \in S} p_{i,j}(u) f^{\mu^*}(j). \tag{3}$$

In Q-learning, $Q(i,u)$ is updated every time an action u is executed in state i. If this execution results in an immediate reward r_{imm} and a transition into state j, then $Q(i,u)$ is updated using the following rule,

$$Q(i,u) \leftarrow Q(i,u) + \beta(r_{imm} + \gamma U_Q(j) - Q(i,u)), \tag{4}$$

where $0 < \gamma < 1$ is the discount factor, $0 < \beta < 1$ is the learning rate, and $U_Q(j)$ is the value of state j, defined as $\max_{a \in U(j)} Q(j,a)$. If the learning algorithm uses an exploration strategy that ensures that it executes each admissible action in each state infinitely often, and the learning rate β is appropriately decayed, Q-learning is guaranteed to converge to an optimal policy [46].

2.2. *The problems of discounting*

Discounted reinforcement learning is well-studied, and methods such as Q-learning and ARTDP are shown to converge under suitable conditions both in theory and in practice. Discounted optimization is motivated by domains where reward can be interpreted as money that can earn interest, or where there is a fixed probability that a run will be terminated at any given time. However, many domains in which we would like to use reinforcement learning do not have either of these properties. As Schwartz pointed out, even researchers who use learning methods that optimize discounted totals in such domains evaluate their systems using a different, but more natural, measure—average expected reward per time step [37]. Discounting in such domains tends to sacrifice bigger long-term rewards in favor of smaller short-term rewards, which is undesirable in many cases. The following example illustrates this.

In the Multi-loop domain shown in Fig. 1, there are four loops of different lengths. The agent has to choose one of the four loops in state S. The average reward per step is $3/3 = 1$ in loop 1, $6/5 = 1.2$ in loop 2, $9/7 = 1.29$ in loop 3, and $12/9 = 1.33$ in loop 4. According to the average-reward optimality criterion, the best policy is to take loop 4, getting the highest average reward of 1.33.

But the discounted optimal policy is different based on the value of the discount factor γ. Let $Reward_i$ be the reward at the end of loop i and $Length_i$ be the total number of steps in loop i. Then, the discounted total reward for the policy μ_i of following loop i is

$$0 + \cdots + \gamma^{Length_i - 1} Reward_i + 0 + \cdots + \gamma^{2Length_i - 1} Reward_i + 0 + \cdots$$

and can be simplified to

$$f^{\mu_i}(S) = \frac{Reward_i \times \gamma^{Length_i - 1}}{1 - \gamma^{Length_i}}.$$

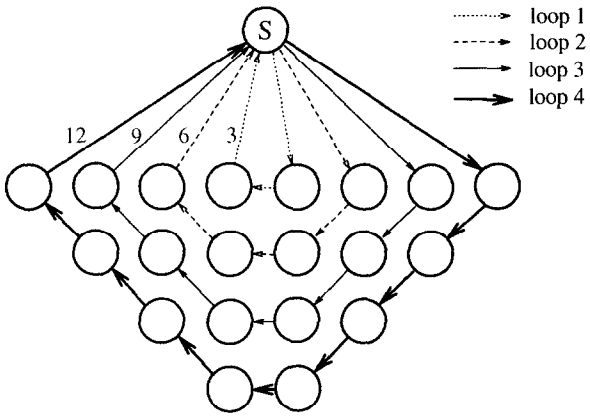

Fig. 1. The Multi-loop domain. Discounted learning methods may converge to a suboptimal policy.

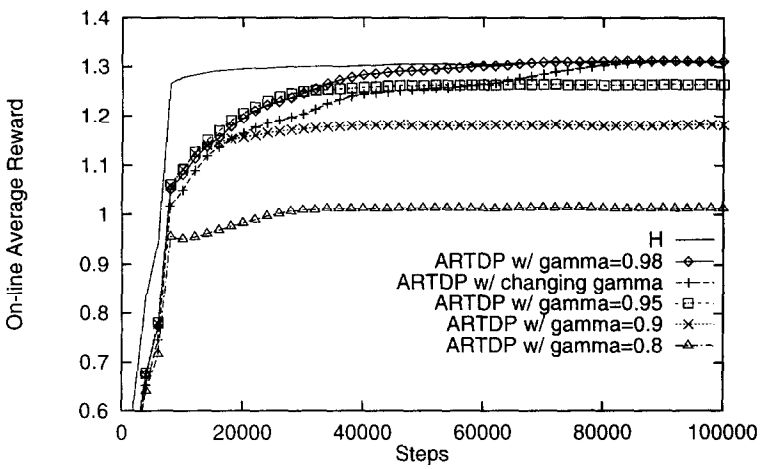

Fig. 2. On-line average reward per step with 10% random exploration in the Multi-loop domain. Each point is the mean of 30 trials over the last 8000 steps.

In particular, the rewards in each loop are such that the optimal discounted policy is to follow loop 1 when $\gamma < 0.85$, loop 2 when $0.85 \leqslant \gamma < 0.94$, loop 3 when $0.94 \leqslant \gamma < 0.97$, and loop 4 when $\gamma \geqslant 0.97$. Hence when γ is greater than 0.97, the policy for optimizing discounting total reward also optimizes the average reward.

Fig. 2 shows the results of running ARTDP in this domain with different values of γ. On the X-axis is the number of steps taken, and on the Y-axis is the on-line average reward for the previous 8000 steps averaged over 30 trials. For comparison, the average reward of our model-based ARL method called H-learning (described in the next section) is computed in the same way and is shown as the solid line. For exploration,

random actions were taken with 0.1 probability in state S. ARTDP converged to loop 1 when γ=0.8, loop 2 when γ=0.9, loop 3 when γ=0.95, and loop 4 when γ=0.98. The H-learning algorithm selected loop 4 in the fewest number of steps (shown with a solid line in Fig. 2). This experimental result confirms that the optimal policy for maximizing discounted total reward is different depending on the value of the discounting factor. When the discount factor is too small, the optimal discounted policy may yield a suboptimal average reward.

As we see in this domain, discounted learning methods can optimize the average reward with a high value of γ ($\geqslant 0.97$). But a high value of γ causes the convergence to be too slow as can be seen in Fig. 2. Since discounted learning methods with a low value of γ need fewer steps to converge than those with a high value of γ, we may expect that they can converge to an optimal average-reward policy faster by starting with a low value of γ and slowly increasing it. To see whether this approach works, the value of γ was gradually increased from 0.8 to 0.98, while running ARTDP. γ was started with 0.8 and was increased to 0.9 when ARTDP converged to loop 1, to 0.95 when it converged to loop 2, and to 0.98 when it converged to loop 3. It was assumed that ARTDP converged to a loop if it selected that loop for a thousand consecutive steps. The result of Fig. 2 shows that changing γ makes ARTDP even slower than when γ is fixed at the highest value, 0.98.

In summary, using discounted learning when the actual optimization criterion is to maximize the gain or average reward leads to short-sighted policies, and to poor performance if the discount factor is low. If the discount factor is high enough, the discounted learning methods can find the optimal average-reward policy; but then they converge too slowly. Moreover, starting from a small γ and slowly increasing it slows down the convergence even further. We will later show that these problems due to discounting naturally arise in real world domains and lead to poor performance of ARTDP and Q-learning in some cases.

3. Average-reward reinforcement learning

We start with the standard Markov Decision Problems (MDP) introduced in Section 2. Recall that $r^{\mu}(s_0, t)$ is a random variable that denotes the total reward received in time t when the agent uses the policy μ starting from s_0. In Average-reward Reinforcement Learning (ARL), we seek to optimize the average expected reward per step over time t as $t \to \infty$. For a given starting state s_0, and policy μ, this is denoted by $\rho^{\mu}(s_0)$ and is defined as

$$\rho^{\mu}(s_0) = \lim_{t \to \infty} \frac{1}{t} E(r^{\mu}(s_0, t)). \tag{5}$$

We say that two states *communicate* under a policy if there is a positive probability of reaching each state from the other using that policy. A *recurrent* set of states is a set of states that communicate with each other and do not communicate with states not in that set. Non-recurrent states are called *transient*. An MDP is *ergodic* if its states form a single recurrent set under each stationary policy. It is a *unichain* if every

stationary policy gives rise to a single recurrent set of states and possibly some transient states [34].

For unichain MDPs the expected long-term average reward per time step for any policy μ is independent of the starting state s_0. We call it the "gain" of the policy μ, denoted by $\rho(\mu)$, and consider the problem of finding a "gain-optimal policy", μ^*, that maximizes $\rho(\mu)$. From now on, unless otherwise specified, whenever we use the term "the optimal policy", we mean the gain-optimal policy.

3.1. Derivation of H-learning

Even though the gain of a policy, $\rho(\mu)$, is independent of the starting state, the total expected reward in time t may not be so. The total reward for a starting state s in time t for a policy μ can be conveniently denoted by $\rho(\mu)t + \varepsilon_t(s)$, where $\varepsilon_t(s)$ is a time-dependent offset. Although $\lim_{t\to\infty} \varepsilon_t(s)$ may not exist for periodic policies, the Cesaro-limit of $\varepsilon_t(s)$, defined as $\lim_{l\to\infty} \frac{1}{l} \sum_{t=1}^{l} \varepsilon_t(s)$, always exists, and is denoted by $h(s)$ [6]. It is called the *bias* of state s and can be interpreted as the expected long-term advantage in total reward for starting in state s over and above $\rho(\mu)t$, the expected total reward in time t on the average.

Suppose the system goes from state i to j using a policy μ. In so doing, it used up a time step that is worth a reward of $\rho(\mu)$ on the average, but gained an immediate reward of $r_i(\mu(i))$. Hence, the bias values of state i and j for the policy μ must satisfy the following equation.

$$h(i) = r_i(\mu(i)) - \rho(\mu) + \sum_{j=1}^{n} p_{i,j}(\mu(i))h(j). \tag{6}$$

The gain-optimal policy μ^* maximizes the right-hand side of the above equation for each state i [6].

Theorem 1 (Howard). *For any MDP, there exist a scalar ρ and a real-valued function h over S that satisfy the recurrence relation*

$$\forall i \in S, \quad h(i) = \max_{u\in U(i)} \left\{ r_i(u) + \sum_{j=1}^{n} p_{i,j}(u)h(j) \right\} - \rho. \tag{7}$$

Further, the optimal policy μ^* attains the above maximum for each state i, and ρ is its gain.

Eq. (7) is the Bellman equation for Average-reward RL problem. Intuitively, this can be explained as follows. In going from a state i to the best next state j, the system gained an immediate reward $r_i(u)$ instead of the average reward ρ. After convergence, if u is the optimal action, the expected long-term advantage for being in state i as opposed to state j must equal the difference between $r_i(u)$ and ρ. Hence the difference between the bias value of state i and the expected bias value of the next state j must equal $r_i(u) - \rho$.

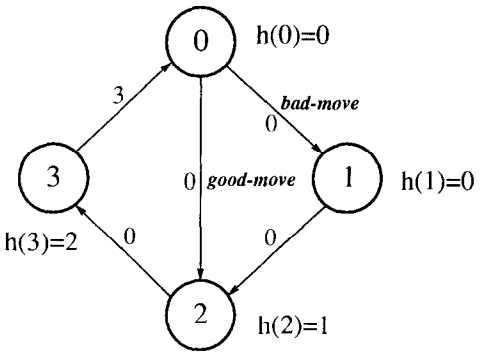

Fig. 3. A simple MDP that illustrates the Bellman equation.

1. 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.
2. $N(i,a) \leftarrow N(i,a) + 1$; $N(i,a,k) \leftarrow N(i,a,k) + 1$
3. $p_{i,k}(a) \leftarrow N(i,a,k)/N(i,a)$
4. $r_i(a) \leftarrow r_i(a) + (r_{imm} - r_i(a))/N(i,a)$
5. $GreedyActions(i) \leftarrow$ All actions $u \in U(i)$ that maximize $\{r_i(u) + \sum_{j=1}^{n} p_{i,j}(u)h(j)\}$
6. If $a \in GreedyActions(i)$, then
   (a) $\rho \leftarrow (1-\alpha)\rho + \alpha(r_i(a) - h(i) + h(k))$
   (b) $\alpha \leftarrow \frac{\alpha}{\alpha+1}$
7. $h(i) \leftarrow \max_{u \in U(i)}\{r_i(u) + \sum_{j=1}^{n} p_{i,j}(u)h(j)\} - \rho$
8. $i \leftarrow k$

Fig. 4. The H-learning algorithm. The agent executes steps 1–8 when in state i.

Notice that any one solution to Eq. (7) yields an infinite number of solutions by adding the same constant to all h-values. However, all these sets of h-values will result in the same set of optimal policies μ^*, since the optimal action in a state is determined only by the relative differences between the values of h. Setting the h-value of an arbitrary recurrent "reference" state to 0, guarantees a unique solution for unichain MDPs.

For example, in Fig. 3, the agent has to select between the actions good-move and bad-move in state 0. For this domain, $\rho = 1$ for the optimal policy of choosing good-move in state 0. If we arbitrarily set $h(0)$ to 0, then $h(1) = 0$, $h(2) = 1$, and $h(3) = 2$ satisfy the recurrence relations in Eq. (7). For example, the difference between $h(3)$ and $h(1)$ is 2, which equals the difference between the immediate reward for the optimal action in state 3 and the optimal average reward 1.

In White's relative value iteration method, the h-value of an arbitrarily chosen reference state is set to 0 and the resulting equations are solved by synchronous successive

approximation [6]. Unfortunately, the asynchronous version of this algorithm that updates ρ using Eq. (7) does not always converge [5]. Hence, instead of using Eq. (7) to solve for ρ, H-learning estimates it from on-line rewards (see Fig. 4).

The algorithm in Fig. 4 is executed in each step, where i is the current state, $N(i,u)$ denotes the number of times u was executed in i, and $N(i,u,j)$ is the number of times it resulted in state j. Our implementation explicitly stores the current greedy policy in the array *GreedyActions*. Before starting, the algorithm initializes α to 1, and all other variables to 0. *GreedyActions* in each state are initialized to the set of admissible actions in that state.

H-learning can be seen as a cross between Schwartz's R-learning [37], which is a model-free average-reward learning method, and Adaptive RTDP (ARTDP) [3], which is a model-based discounted learning method. Like ARTDP, H-learning computes the probabilities $p_{i,j}(a)$ and rewards $r_i(a)$ by straightforward maximum likelihood estimates. It 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

$$h(i) \leftarrow \max_{u \in U(i)} \left\{ r_i(u) + \sum_{j=1}^{n} p_{i,j}(u) h(j) \right\} - \rho. \tag{8}$$

One of the nice properties of H-learning, that is shared by Q-learning and ARTDP, is that it learns the optimal policy no matter what exploration strategy is used during learning, as long as every action of every state is executed sufficiently often. The exploration strategy only effects the speed with which the optimal policy is learned, not the optimality of the learned policy. This is unlike some temporal difference methods such as TD-λ which are designed to learn the value function for the policy that is executed during learning [39].

Just as in Q-learning, in the model-free R-learning Eq. (7) is split into two parts by defining

$$R(i,u) = r_i(u) + \sum_{j=1}^{n} p_{i,j}(u) h(j) - \rho, \tag{9}$$

where

$$h(j) = \max_{u} R(j,u). \tag{10}$$

$R(i,u)$ represents the expected bias value when action u is executed in state i and the gain-optimal policy is followed from then on. Initially all the R-values are set to 0. When action u is executed in state i, the value of $R(i,u)$ is updated using the update equation

$$R(i,u) \leftarrow (1-\beta) R(i,u) + \beta (r_{imm} + h(j) - \rho), \tag{11}$$

where β is the learning rate, r_{imm} is the immediate reward obtained, j is the next state, and ρ is the estimate of the average reward of the current greedy policy. In

any state i, the greedy action u maximizes the value $R(i,u)$; so R-learning does not need to explicitly learn the immediate reward functions $r_i(u)$ or the action models $p_{i,j}(u)$, since it does not need them either for the action selection or for updating the R-values.

As in most RL methods, while using H-learning, the agent makes some exploratory moves—moves that do not necessarily maximize the right-hand side of Eq. (7) and are intended to ensure that every state is visited infinitely often during training. Without exploratory moves, H-learning could converge to a suboptimal policy. However, these moves make the estimation of ρ slightly complicated. Simply averaging the immediate rewards over non-exploratory (greedy) moves would not do, because the exploratory moves could make the system accumulate rewards from states that it never visits if it were always making greedy moves. Instead, we use a method similar to that of R-learning to estimate the average reward [37]. From Eq. (7), in any state i, for any greedy action u that maximizes the right-hand side, $\rho = r_i(u) - h(i) + \sum_{j=1}^{n} p_{i,j}(u)h(j)$. Hence, ρ can be estimated by cumulatively averaging $r_i(u) - h(i) + h(j)$, whenever a greedy action u is executed in state i resulting in state j. Thus, ρ is updated using the following equation, where α is the learning rate,

$$\rho \leftarrow \rho + \alpha(r_i(u) - h(i) + h(j) - \rho). \tag{12}$$

H-learning is very similar to Jalali and Ferguson's Algorithm B [16]. This algorithm was proved to converge to the gain-optimal policy for ergodic MDPs. Since most domains that we are interested in are non-ergodic, to apply this algorithm to such domains, we need to add exploration. Indeed, the role of exploration in H-learning is to transform the original MDP into an ergodic one by making sure that every state is visited infinitely often. Another difference between the two algorithms is that the B-algorithm updates ρ by averaging the immediate reward $r_i(u)$ for each action over the action sequence, rather than averaging the "adjusted immediate reward" $r_{i,j} - h(i) + h(j)$, as we do. Later, we present experimental evidence that shows that this is the crucial reason for the significant difference between the performances of the two algorithms. Thirdly, to make the h-values bounded, Algorithm B chooses an arbitrary recurrent reference state and permanently grounds its h-value to 0. We found that this change slows down H-learning in many cases. To extend the proof of convergence of Algorithm B to our case, we have to show that the above changes preserve its correctness under suitable conditions of exploration. Unfortunately, the original proof of convergence is quite involved and its correctness is disputed.[3] In any case, it may be easier to give an independent convergence proof based on stochastic approximation theory.

3.2. AGV Scheduling

Automatic Guided vehicles (AGVs) are used in modern manufacturing plants to transport materials from one location to another [27]. To compare the performance of various learning algorithms, a small AGV domain called the "Delivery domain" shown in

[3] D. Bertsekas, private communication.

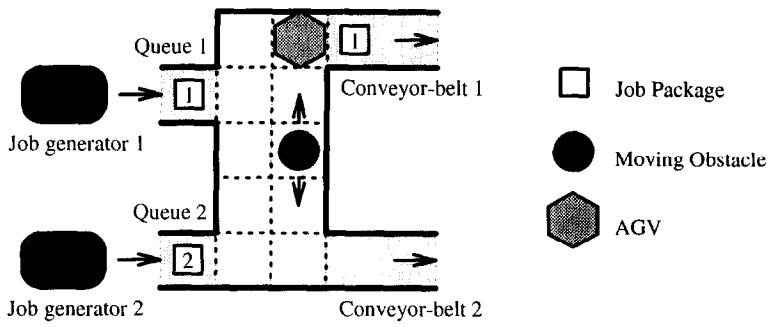

Fig. 5. The Delivery domain.

Fig. 5 was used. There are two job generators on the left, one AGV, and two destination conveyor belts on the right. Each job generator produces jobs and puts them on its queue as soon as it is empty. The AGV loads and carries a single job at a time to its destination conveyor belt. Each job generator can generate either a type 1 job with K units of reward when delivered to belt 1, or a type 2 job with 1 unit of reward when delivered to belt 2. The probability of generating job 1 is p for generator 1, and q for generator 2.

The AGV moves on two lanes of 5 positions each, and can take one of six actions at a time: do-nothing, load, move-up, move-down, change-lane, and unload. To load a job, the AGV must be in the position next to the queue. To unload a job, it must be next to the proper conveyor belt. To make this domain more interesting, a moving obstacle is added. It randomly moves up or down in each instant, but can only stay in the right lane and cannot stand still. The AGV and the obstacle can both move in a single time step. If the obstacle collides with the AGV when the AGV is delivering a job or is standing still, the state remains unchanged. There is a penalty of -5 for all collisions with the obstacle.

A state is specified by the two job numbers in the queues, the locations (X-Y coordinates) of the AGV and the obstacle, and the job number on the AGV. We assume that each queue can hold a single job and the complete state is observable to the learning system. There are a total of 540 different states in this domain. The goal of the AGV is to maximize the average reward received per unit time, i.e., find the gain-optimal policy.

By varying the reward ratio of the jobs and/or the job mixes produced by the job generators, the optimal policy is changed. For example, when $K = 1$, and both the job generators produce type 1 jobs with very low rates p and q, the AGV should unload jobs from queue 2 much more frequently than from queue 1 because the number of time steps needed to transport type 2 jobs from queue 2 to belt 2 is much smaller than that needed to move them from queue 1 to belt 2. But, when both the job generators produce jobs of type 1 with a high rate, and $K = 5$, the AGV should unload jobs from queue 1 much more frequently than from queue 2, because the increased value of job 1 more than compensates for the extra distance. It is, in general, hard to predict the best policy given different values of p, q, and K.

3.3. Experimental results

Our experiments are based on comparing H-learning with ARTDP, Q-learning, R-learning, and the B-algorithm of Jalali and Ferguson in the Delivery domain.

For these experiments, the Delivery domain is used with $p = 0.5$, and $q = 0.0$. In other words, generator 1 produces both types of jobs with equal probability, while generator 2 always produces type 2 jobs. We present the result of comparing H-learning with ARTDP, Q-learning, and R-learning in two situations of the AGV domain: $K = 1$ and $K = 5$. We chose these two sets of domain parameters because they illustrate two qualitatively different situations. Experiments on a wider range of domain parameters are reported elsewhere [31].

Each experiment was repeated for 30 trials for each algorithm. Every trial started from a random initial state. In all our experiments, a random exploration strategy was used, in which with a probability $1 - \eta$, a greedy action is chosen, and with a probability $\eta = 0.1$, an action is chosen uniformly randomly over all available actions. While training, the average reward per step is computed over the last 10,000 steps for $K = 1$, and over the last 40,000 steps for $K = 5$. H-learning and the B-algorithm do not have any parameters to tune. The parameters for the other learning methods are tuned by trial and error to get the best performance. Strictly speaking, γ is part of the problem definition of discounted optimization, and not a parameter of the learning algorithm. However, since our goal in this domain is to see how well the discounted methods can approach learning a gain-optimal policy, we treated it as an adjustable parameter. For $K = 1$ case, the only parameter for ARTDP is $\gamma = 0.9$, the parameters for Q-learning are $\beta = 0.05$ and $\gamma = 0.9$, and for R-learning, $\beta = 0.01$, $\alpha = 0.05$. For $K = 5$ case, for ARTDP $\gamma = 0.99$, for Q-learning $\beta = 0.05$ and $\gamma = 0.99$, and for R-learning, $\beta = 0.01$, $\alpha = 0.005$. The results are shown in Fig. 6.

When $K = 1$, since both jobs have the same reward, the gain-optimal policy is to always serve generator 2 that produces only type 2 jobs. Since the destination of these jobs is closer to their generator than type 1 jobs, it is also a discounted optimal policy. We call this type of domains "short-range domains" where the discounted optimal policy for a small value of γ coincides with the gain-optimal policy. For this parameter setting of the delivery domain, serving queue 2 is the short-term optimal policy as well as the long-term optimal policy. In this case, the model-based discounted method, ARTDP, converges to the gain-optimal policy slightly faster than H-learning, although the difference is negligible. All methods except R-learning show almost the same performance.

When K is set to 5, the AGV receives five times more reward by unloading the jobs of type 1 than the jobs of type 2. The gain-optimal policy here is to serve the jobs from queue 1 all of the time except when both of the queues have type 2 jobs and the obstacle is located near conveyor-belt 1. This policy conflicts with the discounted optimal policy when $\gamma = 0.9$. Even when the AGV serves the generator 1, because it also generates jobs for belt 2, it often has to go there. Whenever it is close to belt 2, ARTDP sees a short-term opportunity in serving generator 2 and does not return to generator 1, thus failing to transport high reward jobs. Hence it cannot find the gain-optimal policy when $\gamma = 0.9$. To overcome this difficulty, γ is set to

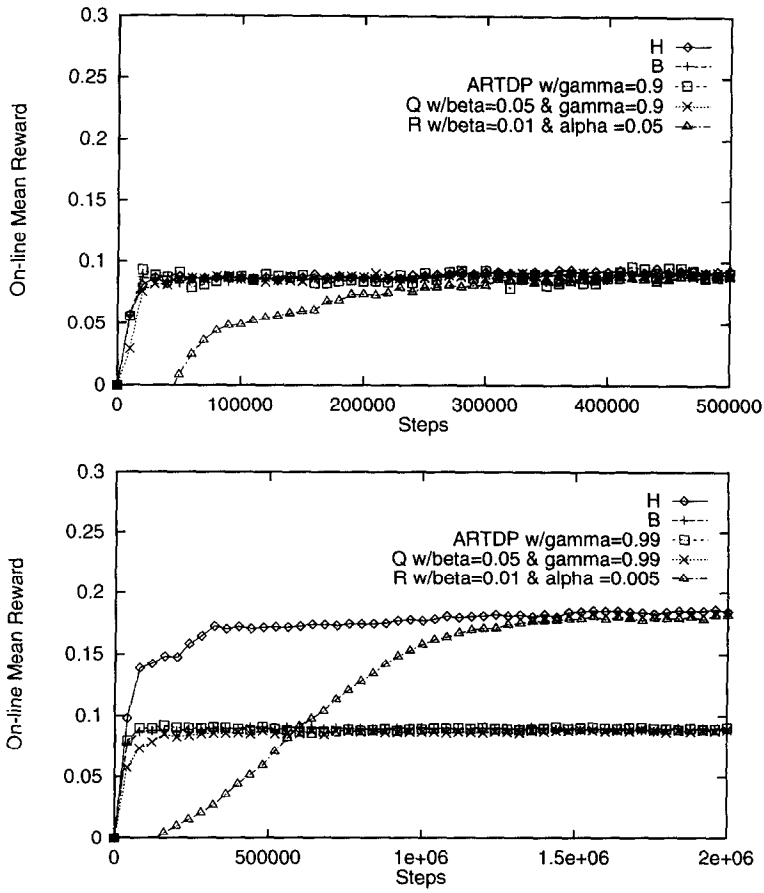

Fig. 6. Average reward per step for H-learning, B-algorithm, ARTDP, Q-learning, and R-learning in the Delivery domain estimated over 30 trials with random exploration with $\eta = 0.1$. Top: $p = 0.5$, $q = 0.0$, and $K = 1$. Average reward is estimated over the last 10,000 steps. Bottom: $p = 0.5$, $q = 0.0$, and $K = 5$. Average reward is estimated over the last 40,000 steps.

0.99. With this value of γ, the discounted optimal policy is the same as the gain-optimal policy. Even so, the discounted learning methods, ARTDP and Q-learning, as well as the undiscounted B-algorithm of Jalali and Ferguson could not find the gain-optimal policy in 2 million steps. Since the discount factor γ=0.99 is high, the discounted methods need longer training time or higher exploration rate to learn the optimal policy. As we can infer from Fig. 6, in this "long-range" domain, ARTDP, Q-learning, and the B-algorithm served queue 2 exclusively for all trials getting a gain less than 0.1, while H-learning and R-learning were able to find a policy of gain higher than 0.18.

Thus the average-reward learning methods, H-learning and R-learning, significantly outperformed the discounted learning methods, ARTDP and Q-learning, in finding the

gain-optimal policy. H-learning and R-learning served both queues for all 30 trials. But, R-learning took more training steps to converge than H-learning and learned a policy of less gain than H-learning did. Somewhat surprisingly, the B-algorithm, which is designed to optimize the average reward, is also unable to find the gain-optimal policy. Since we also allowed the B-algorithm to do random exploration and prevented it from grounding the h-value of a reference state to 0, the only difference between this version of the B-algorithm and the H-learning is the way ρ is updated. The poor performance of the B-algorithm suggests that it is crucial to adjust the immediate rewards using the Eq. 12 while updating ρ.

The main reason that ARTDP could not find the gain-optimal policy when $K = 5$, even though it is also the discounted optimal policy when $\gamma = 0.99$, is that a high value of γ reduces the effect of discounting and makes the temporally far off rewards relevant for optimal action selection. Since it takes a long time to propagate these rewards back to the initial steps, it takes a long time for the discounted methods to converge to the true optimum. Meanwhile the short-term rewards still dominate in selecting the action and result in low average reward.

The reason that model-based learning methods converge in fewer steps than model-free learning methods is that they propagate more information in each step by taking the expectation over all the possible next states for a given action in each update. This also requires the model-based learning methods to learn and store the action models explicitly, and increases the CPU-time for each update. So, we compared the performance of H-learning with that of ARTDP, Q-learning, and R-learning as a function of CPU time. Fig. 7 shows these results. Each point is the on-line average reward of 30 trials over the last 40,000 steps with random exploration with $\eta = 0.1$. All parameters for learning methods are the same as those in Fig. 6.

When $K = 1$, Q-learning converged to the gain-optimal policy in the shortest time. R-learning was the slowest. H-learning and ARTDP showed almost the same performance. When $K = 5$, the discounted learning methods, ARTDP and Q-learning, could not converge to the gain-optimal policy, whereas the two average-reward learning methods, H-learning and R-learning, did. Even though H-learning had good performance in the beginning, R-learning converged to the gain-optimal policy slightly faster than H-learning.

The results of this experiment show that in short-range domains where discounted optimal policy coincides with the gain-optimal policy, H-learning performs as well as ARTDP and Q-learning, and better than R-learning with respect to number of steps. However, the model-free methods show slightly better performance with respect to CPU time. But in long-range domains where discounted optimal policy conflicts with the gain-optimal policy, discounted methods such as ARTDP and Q-learning either take too long to converge to the gain-optimal policy or, if γ is low, converge to a policy with less gain. H-learning achieves higher average reward in fewer steps than the other methods in such cases with no parameter tuning.

In a different experiment, we tested the robustness of H-learning with respect to changes in the domain parameters p, q, and K in the Delivery domain. We experimented with a total of 75 different domain parameter settings, by varying p, q, and K. H-learning was compared to ARTDP with $\gamma = 0.9$, 0.99, and 0.999.

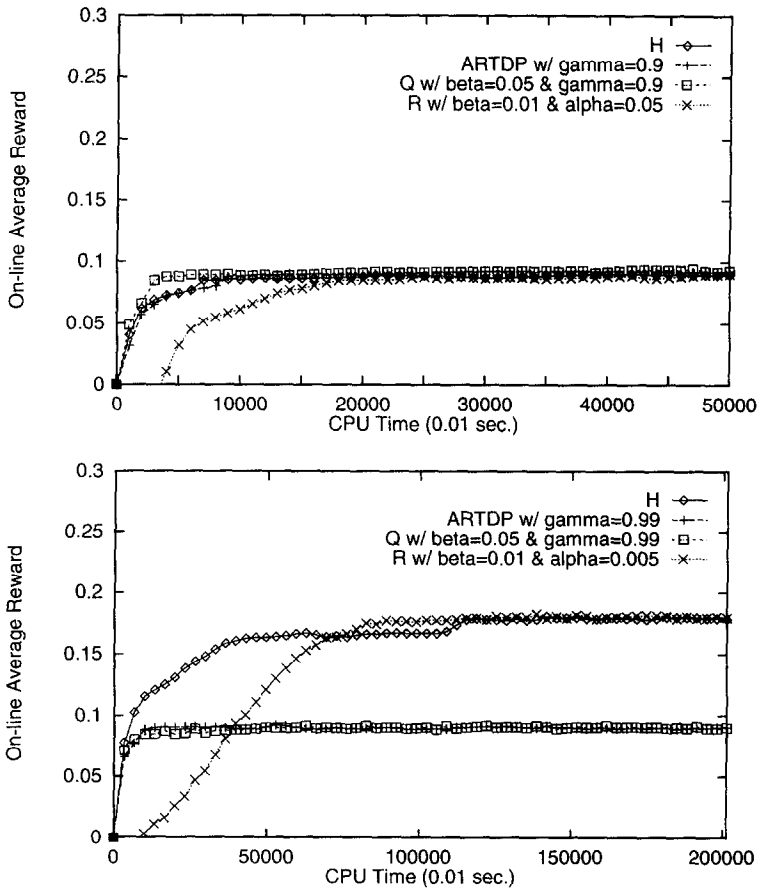

Fig. 7. Average rewards per step plotted against CPU time for H-learning, ARTDP, Q-learning, and R-learning in the Delivery domain estimated over 30 trials with random exploration with $\eta = 0.1$. Top: $p = 0.5$, $q = 0.0$, and $K = 1$. Average reward is estimated over the last 10 seconds of CPU time. Bottom: $p = 0.5$, $q = 0.0$, and $K = 5$. Average reward is estimated over the last 40 seconds of CPU time.

H-learning was able to find the optimal policy in all 75 cases, whereas ARTDP with the best γ value (0.99) was able to find the optimal policy in 61 cases. We found that these 75 different configurations of the domain can be roughly divided into two groups—50 configurations can be considered "short-range". In these configurations, ARTDP with $\gamma = 0.9$ and H-learning were comparable. They both found the gain-optimal policy, and ARTDP sometimes found it in slightly fewer steps than H-learning. In the remaining 25 "long-range" configurations, however, ARTDP with $\gamma = 0.9$ could not find the gain-optimal policy. Increasing γ to 0.99 helped ARTDP find the gain-optimal policy, but much more slowly than H-learning. Increasing γ to 0.999, however, decreased the success rate of ARTDP (in 300,000 steps), because it slowed down the convergence too drastically [31].

In summary, our experiments indicate that H-learning is more robust with respect to changes in the domain parameters, and in many cases, converges in fewer steps to the gain-optimal policy than its discounted counterpart. Thus our experiments suggest that if our goal is to find the gain-optimal policies, then Average-reward Reinforcement Learning methods are preferable to the discounted methods. Our results are consistent with those of Mahadevan who compared Q-learning and R-learning in a robot simulator domain and a maze domain and found that R-learning can be tuned to perform better [24].

4. Exploration

Recall that H-learning needs exploratory actions to ensure that every state is visited infinitely often during training in order to avoid converging to suboptimal policies. Unfortunately, actions executed exclusively for exploratory purpose could lead to decreased reward, because they do not fully exploit the agent's current knowledge of how to maximize its reward.

In this section, we will describe a version of H-learning called Auto-exploratory H-learning (AH), which automatically explores the promising parts of the state space while always executing current greedy actions. Our approach is based on the idea of "optimism under uncertainty", and is similar to Kaelbling's Interval Estimation (IE) algorithm, and Koenig and Simmons's method of representing the reward functions using action-penalty scheme [17, 18, 20].

4.1. Auto-exploratory H-Learning

Recall that in ergodic MDPs, every stationary policy is guaranteed to visit all states. In these MDPs, it can be shown that always choosing a greedy action with respect to the current value function ensures sufficient exploration, although a better exploration strategy might speed up convergence even further. Hence, we are primarily interested in non-ergodic domains in this section. Unfortunately, the gain of a stationary policy for a general multichain (non-unichain) MDP is not constant but depends on the initial state [34]. Hence we consider some restricted classes of MDPs. An MDP is *communicating* if for every pair of states i, j, there is a stationary policy under which they communicate. Contrast this with ergodic MDPs, where every pair of states communicate under *every* stationary policy. For example, our Delivery domain is communicating, but not ergodic. Serving only one of the generators all the time prevents the AGV from visiting some states. A *weakly communicating* MDP is more general than a communicating MDP and also allows a set of states which are transient under every stationary policy [34]. Although the gain of a stationary policy for a weakly communicating MDP also depends on the initial state, the gain of an optimal policy does not. AH-learning exploits this fact, and works by using ρ as an upper bound on the optimal gain. It does this by initializing ρ to a high value and by slowly reducing it to the gain of the optimal policy. Instead of as an estimate of average reward of the current greedy policy, we now interpret ρ as the aspired average reward of the learner. The aspired average reward decreases slowly,

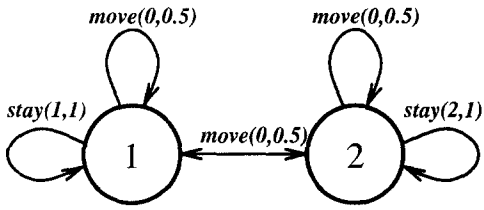

Fig. 8. The Two-state domain. The notation action(r, p) on the arc from a node 1 to 2 indicates that, when action is executed from 1, p is the probability of the next state being 2 and r is the immediate reward.

while the actual average reward of the current greedy policy increases, and the difference between them decreases with time. When the aspired value is the same as the average reward of the current greedy policy, AH-learning converges. To ensure that AH-learning converges to a globally optimal policy, we have to adjust the initial value of the aspired average reward and its learning rate so that it never falls below the average reward of any suboptimal greedy policy. AH-learning is applicable to find gain-optimal policies for weakly communicating MDPs, a strict superset of unichains.

There are two reasons why H-learning needs exploration: to learn accurate action and reward models, and to learn correct h values. Inadequate exploration could adversely affect the accuracy of either of these, making the system converge to a suboptimal policy.

The key observation in the design of AH-learning is that the current value of ρ affects how the h-values are updated for the states in the current greedy policy. Let μ be the current suboptimal greedy policy, and $\rho(\mu)$ be its gain. Consider what happens if the current value of ρ is less than $\rho(\mu)$. Recall that $h(i)$ is updated to be $\max_{u \in U(i)}\{r_i(u) + \sum_{j=1}^{n} p_{i,j}(u)h(j)\} - \rho$. Ignoring the changes to ρ itself, the h-values for states in the current greedy policy tend to increase on the average, because the sum of immediate rewards for this policy in any n steps is likely to be higher than $n\rho$ (since $\rho < \rho(\mu)$). It is possible, under these circumstances, that the h-values of *all* states in the current policy increase or stay the same. Since the h-values of states not visited by this policy do not change, this implies that by executing the greedy policy, the system may never be able to get out of the set of states in the current greedy policy. If the optimal policy involves going through states not visited by the greedy policy, it will never be learned.

This is illustrated clearly in the Two-state MDP in Fig. 8, which is a communicating multichain. In each of the two states, there are two actions available: stay and move. stay always keeps the system in the same state as it is currently in; but move changes it with 50% probability. There is no immediate reward for the move action in either state. There is a reward of 1 for the stay action in state 1 and a reward of 2 for the stay action in state 2. In this domain, the optimal policy μ^* is taking the action move in state 1 and stay in state 2 with $\rho(\mu^*) = 2$.

When actions are always chosen greedily, H-learning finds the gain-optimal policy in approximately half of the trials for this domain—those trials in which the stay action in state 2 is executed before the stay action in state 1. If the stay action in state 1 is executed before that in state 2, it receives a reward of $+1$ and updates $h(1)$ to $1 + h(1) - \rho$.

The greedy policy μ is to continue to execute stay in state 1. If $\rho < \rho(\mu) = 1$, this increases the value of $h(1)$ in every update until finally ρ converges to 1. Since greedy action choice always results in the stay action in state 1, H-learning never visits state 2 and therefore converges to a suboptimal policy.

Now consider what happens if $\rho > \rho(\mu)$, where μ is a current greedy policy. In this case, by the same argument as before, the h-values of the states in the current greedy policy must *decrease* on the average. This means that eventually the states outside the set of states visited by the greedy policy will have their h-values higher than some of those visited by the greedy policy. Since the MDP is assumed to be weakly communicating, the recurrent states with higher h-values are reachable from the states with decreasing h-values, and eventually will be visited. Thus, ignoring the transient states that do not affect the gain, as long as $\rho > \rho(\mu)$, there is no danger of getting stuck in a suboptimal policy μ. This suggests changing H-learning so that it starts with a high initial ρ-value, ρ_0, which is high enough so that it never gets below the gain of any suboptimal policy.

In the preceding discussion, we ignored the changes to the ρ-value itself. In fact, ρ is constantly changing at a rate determined by α. Hence, even though ρ was initially higher than $\rho(\mu)$, because it decreases continuously, it can become smaller than $\rho(\mu)$ after a while. To make the previous argument work, we have to adjust α so that ρ changes slowly compared to the h-values. This can be done by starting with a sufficiently low initial α-value, α_0, and decaying it gradually. We denote H-learning with the initial values ρ_0 and α_0 by $\mathrm{H}^{\rho_0,\alpha_0}$. Hence, the H-learning we used until now is $\mathrm{H}^{0,1}$.

So far, we have considered the effect of lack of exploration on the h-values. We now turn to its effect on the accuracy of action models. For the rest of the discussion, it is useful to define the utility $R(i,a)$ of a state action pair (i,a) to be

$$R(i,a) = r_i(a) + \sum_{j=1}^{n} p_{i,j}(a)h(j) - \rho. \tag{13}$$

Hence, the greedy actions in state i are actions that maximize the R-value in state i.

Consider the following run of $\mathrm{H}^{6,0.2}$ in the Two-state domain, where, in step 1, the agent executes the action stay in state 1. It reduces $h(1) = R(1,\text{stay})$ to $1-\rho$ and takes the action move in the next step. Assume that move takes it to state 1 because it has 50% failure rate. With this limited experience, the system assumes that both the actions have the same next state in state 1, and stay has a reward of 1 while move has 0. Hence, it determines that $R(1,\text{stay}) = 1+h(1)-\rho > 0+h(1)-\rho = R(1,\text{move})$ and continues to execute stay, and keeps decreasing the value of $h(1)$. Even though $h(2) > h(1)$, the agent cannot get to state 2 because it does not have the correct action model for move. Therefore, it keeps executing stay, which in turn makes it impossible to learn the correct model of move. Unfortunately, this problem cannot be fixed by changing ρ_0 or α_0.

The solution we have implemented, called "Auto-exploratory H-Learning" (AH-learning), starts with a high ρ_0 and low α_0 ($\mathrm{AH}^{\rho_0,\alpha_0}$), and stores the R-values explicitly. In H-learning, all R-values of the same state are effectively updated at the same time by updating the h-value, which sometimes makes it converge to incorrect action models. In AH-learning, $R(i,a)$ is updated by the following update equation (cf. Eq. (13)) only when action a is taken in state i.

1. Take a greedy action, i.e., an action $a \in U(i)$ that maximizes $R(i,a)$ in the current state i. Let k be the resulting state, and r_{imm} be the immediate reward received.
2. $N(i,a) \leftarrow N(i,a) + 1$; $N(i,a,k) \leftarrow N(i,a,k) + 1$
3. $p_{i,k}(a) \leftarrow N(i,a,k)/N(i,a)$
4. $r_i(a) \leftarrow r_i(a) + (r_{imm} - r_i(a))/N(i,a)$
5. $\rho \leftarrow (1-\alpha)\rho + \alpha r_i(a)$
6. $\alpha \leftarrow \frac{\alpha}{\alpha+1}$
7. $R(i,a) \leftarrow r_i(a) + \sum_{j=1}^{n}\{p_{i,j}(a)h(j)\} - \rho$, where $h(j) = \max_u R(j,u)$.
8. $i \leftarrow k$

Fig. 9. The AH-learning algorithm. The agent executes steps 1–8 when in state i.

$$R(i,a) \leftarrow r_i(a) + \sum_{j=1}^{n} p_{i,j}(a)h(j) - \rho. \tag{14}$$

In AH-learning, when ρ is higher than the gain of the current greedy policy, the R-value of the executed action is decreased, while the R-values of the other actions remain the same. Therefore, eventually, the un-executed actions appear to be the best in the current state, forcing the system to explore such actions. Thus, AH-learning is forced to execute all actions, learn correct models and find the optimal policy by executing only greedy actions.

The algorithm for AH-learning is shown in Fig. 9. ρ and α are initially set to user-defined parameters ρ_0 and α_0. Unlike H-learning, our implementation of AH-learning does not explicitly store the current greedy policy. There is also no need to check if an executed action is greedy before updating ρ, since all executed actions are greedy. ρ can be updated simply by taking the average with the immediate reward, since there are no non-greedy actions that distort this estimate. The R-values are stored explicitly rather than the h-values, and are updated by Eq. (14).

Fig. 10(a) shows the plot of R-values from a single run of $AH^{6,0.2}$ in the Two-state domain. All initial R-values are 0. Because the immediate reward of any action is much lower than the initial value of ρ, updating of R-values rapidly reduces them in the beginning. Thus, the action just executed is rarely chosen as the best action the next time. Therefore, $AH^{6,0.2}$ can learn accurate action models by executing each action in each state many times. As ρ gets close to 2, $R(2, \text{stay})$ reduces very little, because the "error measure", $r(2, \text{stay}) - \rho$ is almost 0. But the R-values for other actions will be decreased significantly whenever they are executed, because ρ is significantly higher than their immediate reward. Thus, the system finds the gain-optimal policy in the Two-state domain.

Fig. 10(b) shows the on-line average reward for 100 trials of $AH^{6,0.2}$, $H^{6,0.2}$, and $H^{0,1}$ in the Two-state domain shown in Fig. 8. When actions are chosen greedily, $AH^{6,0.2}$ found the optimal policy in all 100 trials tested, whereas $H^{0,1}$, and $H^{6,0.2}$ found the optimal policy in 57 and 69 trials respectively. This confirms our hypothesis that AH-learning explores the search space effectively while always executing greedy actions. On this very simple Two-state domain, all previously discussed learning methods except AH-

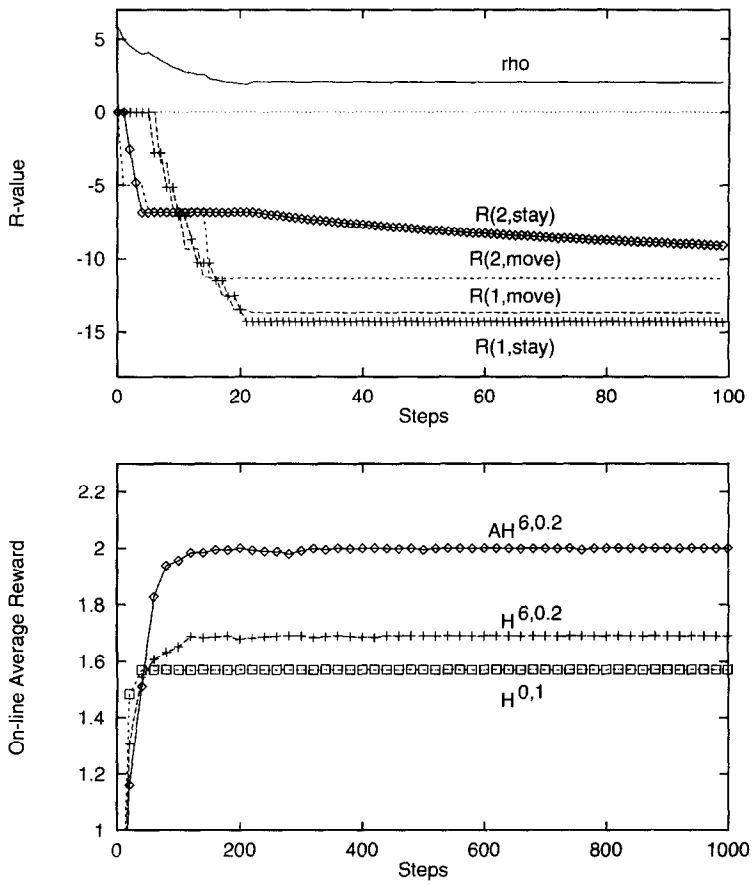

Fig. 10. (a) The R values and ρ values of $AH^{6,0.2}$ in a single trial (top), and (b) the on-line mean rewards of last 20 steps of $AH^{6,0.2}$, $H^{6,0.2}$, and $H^{0,1}$ mean-averaged over 100 trials (bottom) in the Two-State domain.

learning, including H-learning, ARTDP, Q-learning, and R-learning need to occasionally execute non-greedy actions to find the gain-optimal policy.

From our empirical results and informal reasoning, we conjecture that if ρ is maintained higher than the gain of any suboptimal policy during learning, by initializing it to a sufficiently high value and α to a sufficiently small value, then AH-learning converges to the gain-optimal policy. However, if ρ becomes lower than the gain of some suboptimal policy any time during learning, AH-learning might converge to that policy.

4.2. *Experimental results on AH-learning*

In this section, we present more empirical evidence to illustrate the effectiveness of AH-learning in finding a policy with a higher average reward, and in converging quickly

when compared to other previously studied exploration methods. We use the Delivery domain of Fig. 5 to do this.

We compared AH-learning to four other exploration methods: random exploration, counter-based exploration, Boltzmann exploration, and recency-based exploration [45]. In random exploration, a random action is selected uniformly from among the admissible actions with a small probability η. With a high probability $1 - \eta$, in any state i, a greedy action, i.e., one that maximizes $R(i, a)$ over all a, is chosen. In counter-based exploration, an action a is chosen that maximizes

$$R(i,a) + \delta \frac{c(i)}{\sum_{j=1}^{n} P_{i,j}(u)c(j)},$$

where $c(i)$ is the number of times state i is visited, and δ is a small positive constant. In Boltzmann exploration, an action a is selected in state i with probability

$$\frac{e^{R(i,a)/\beta}}{\sum_u e^{R(i,u)/\beta}},$$

where β is the temperature or randomness parameter. In recency-based exploration, an action a is selected that maximizes $R(i, a) + \varepsilon\sqrt{n(i, a)}$, where $n(i, a)$ is the number of steps since the action a is executed in state i last, and ε is a small positive constant. In all the four cases, the parameters η, δ, β, and ε were decayed at a constant rate. Their initial values and the rates of decay were tuned by trial and error to give the best performance.

The parameters for the Delivery domain, p, q and K, were set to 0.5, 0.0 and 5 as in Fig. 6(b). Proper exploration is particularly important in this domain for the following reasons: first, the domain is stochastic; second, it takes many steps to propagate high rewards; and third, there are many suboptimal policies with gain close to the optimal gain. For all these reasons, it is difficult to maintain ρ consistently higher than the gain of any suboptimal policy, which is important for AH-learning to find the gain-optimal policy. It gave the best performance with $\rho_0 = 2$ and $\alpha_0 = 0.0002$.

Fig. 11 shows the on-line average rewards over 30 trials of $AH^{2,0.0002}$, $H^{1,0.001}$, and $H^{0,1}$, with only greedy actions, and of $H^{0,1}$ with 4 different exploration methods: random exploration with an initial $\eta = 0.14$ that is decayed by 0.00031 every 1000 steps; counter-based exploration with an initial $\delta = 0.07$ that is decayed at the rate of 0.00021 every 1000 steps; Boltzmann exploration with an initial $\beta = 0.3$ that is decreased by 0.0003 every 1000 steps; and recency-based exploration with an initial $\varepsilon = 0.05$ that is reduced by 0.0004 every 1000 steps.

When actions are always greedily chosen, $H^{0,1}$ could not find the optimal policy even once. By proper tuning of ρ_0 and α_0, it improved significantly, and was only slightly worse than AH and recency-based methods. Recency-based exploration appears much better than random exploration for this domain, while counter-based and Boltzmann explorations seem worse. AH-learning is faster than other exploration methods, although it dives down to a very low value in the very beginning, which can be attributed to its optimism under uncertainty.

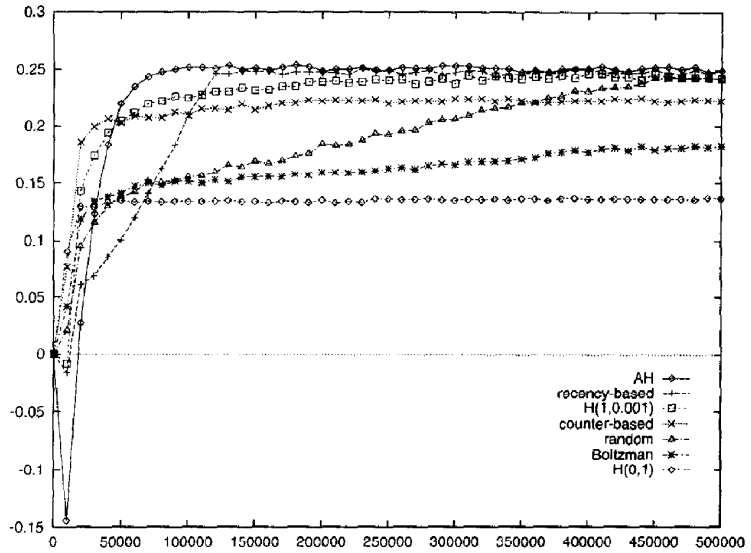

Fig. 11. The on-line mean rewards of last 10K steps averaged over 30 trials for $AH^{2,0.0002}$, $H^{1,0.001}$, and $H^{0,1}$ with only greedy action-selection, and for $H^{0,1}$ with recency-based, counter-based, random, and Boltzman exploration strategies in the Delivery domain with $p = 0.5$, $q = 0.0$, and $K = 5$.

It is interesting to compare AH-learning with other RL methods with respect to CPU time. The performance curves of all methods other than AH-learning in Fig. 12 were copied from Fig 7. We tested AH-learning with $K = 1$ and 5, and added these results to Fig. 12. Since all methods other than AH-learning used un-decayed random exploration with $\eta = 0.1$, their final on-line average rewards were significantly lower than that of AH-learning. Perhaps more importantly, AH-learning converged to the optimal policy in much less CPU time than the other methods in the long-range domain. In particular, unlike H-learning, AH-learning was significantly faster than R-learning.

These results suggest that with proper initialization of ρ and α, AH-learning explores the state space much more effectively than the other exploration schemes. Although AH-learning does involve tuning two parameters ρ and α, it appears that at least ρ can be automatically adjusted. One way to do this is to keep track of the currently known maximum immediate reward over all state-action pairs, i.e., $\max_{i,u} R_i(u)$, and reinitialize ρ to something higher than this value whenever it changes.

5. Scaling-up average-reward reinforcement learning

All table-based RL methods including H-learning and AH-learning are based on dynamic programming and need space proportional to the number of states to store the value function. For interesting real-world domains, the number of states can be enormous, causing a combinatorial explosion in the time and space requirements for these algorithms. In fact, all table-based RL methods need space exponential in the

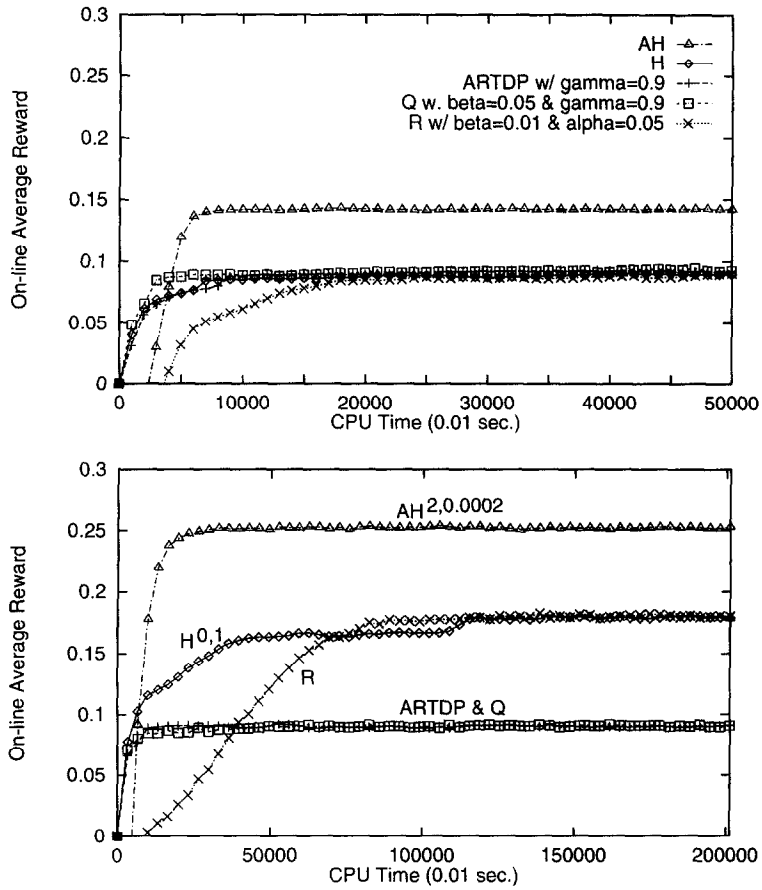

Fig. 12. (a) Average rewards versus training time for $AH^{1.0.0005}$, $H^{0.1}$, ARTDP, Q-learning, and R-learning with $K = 1$ (top), and (b) average rewards versus training time for $AH^{2.0.0002}$, $H^{0.1}$, ARTDP, Q-learning, and R-learning with $K = 5$ (bottom). Each point is the mean of the average reward over the last 40,000 steps for 30 trials with $p = 0.5$ and $q = 0.0$. All methods except AH-learning use random exploration with $\eta = 0.1$.

number of state variables (number of machines, jobs to be transported, number of AGVs, etc.) just to store the value function. In addition, table-based algorithms completely compartmentalize the values of individual states. If they learn the best action in a particular state, it has absolutely no influence on the action they choose in a similar state. In realistic domains, the state spaces are so huge that an agent can never expect to have enough experience with each state to learn the appropriate action. Thus, it is important to generalize to states that have not been experienced by the system, from similar states that have been experienced. This is usually done by finding an approximation for the value function from a hypothesized space of functions.

There have been several function approximation methods studied in the discounted RL literature, including neural network learning [8,21], clustering [26], memory-based methods [28], and locally weighted regression [29,36]. Two characteristics of the AGV

scheduling domain attracted us to local linear regression as the method of choice. First, the location of the AGV is one of the most important features of the state in this domain. Any function approximation scheme must be able to generalize specific locations of the AGV to large regions. Second, the value function for H-learning varies linearly over a region of state space, when the optimal action and its effects, i.e., the immediate reward and the changes in the feature values of the current state, are constant throughout this region. In the AGV domain, this implies that the value function is piecewise linear in the X- and Y- coordinates of the AGV. This is true because the optimal action is the same in large geometrically contiguous regions, it changes only the coordinates of the AGV and by a constant amount, and it gives a constant immediate reward (usually 0).

While model-free learning methods need only to store the value function for each state-action pair, model-based learning methods such as H-learning also need space to store the domain model, which is the combination of the action models and reward models as defined in Section 2. The space requirement for storing the domain model is also exponential in the number of state variables. Dynamic Bayesian networks have been successfully used in the past to represent the domain models [12,35]. In many cases, it is possible to design these networks in such a way that a small number of parameters are sufficient to fully specify the domain models.

5.1. Model generalization using Bayesian networks

One of the disadvantages of model-based methods like H-learning is that explicitly storing its action and reward models consumes a lot of space. The space requirement for storing the models can be anywhere from $O(nm)$ to $O(n^2m)$ depending on the stochasticity of the domain, where n is the number of states and m is the number of actions. To scale the model-based learning methods to large domains, we represent the domain models using dynamic Bayesian networks. In this section, we describe how we can adapt H-learning to learn the parameters for them, assuming that the network structure is given.

We assume that a state is described by a set of discrete valued features. A dynamic Bayesian network (DBN) represents the relationships between the feature values and the action at time t, and the feature values at time $t+1$. A Bayesian network is a directed acyclic graph whose nodes represent random variables, along with a conditional probability table (CPT) associated with every node. The CPT at each node describes the probabilities of different values for a node conditioned on the values of its parents. The probability of any event given some evidence is determined by the network and the associated CPTs, and there are many algorithms to compute this [35]. Since the network structure is given as prior knowledge, learning action models reduces to learning the CPTs.

We illustrate the dynamic Bayesian network representation using the Slippery-lane domain in Fig. 13(a). There is a job generator on the left and a conveyor-belt on the right. The job generator generates a new job immediately after the AGV loads a job. The goal of the AGV is to repeatedly load a job in the loading zone and unload it in the unloading zone. A state of this domain can be described by the job on AGV (Job-on-AGV (which is 0 or 1) and the AGV location (AGV-Loc).

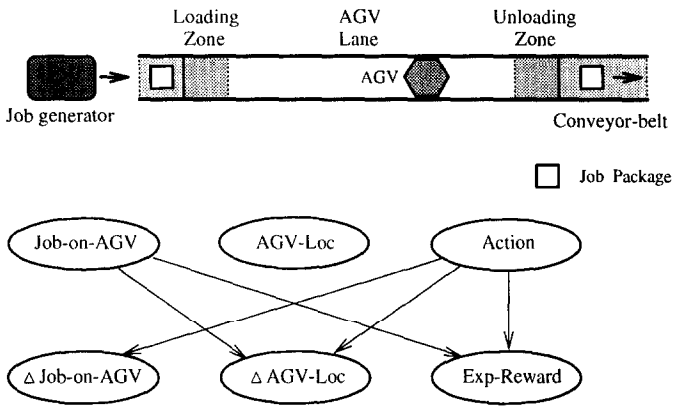

Fig. 13. (a) The Slippery-lane domain (top) and (b) its dynamic Bayesian network (bottom).

There are six actions: stay, load, unload, forward, backward, and move. The action forward is for entering the lane from the loading zone, and backward is for entering the lane from the unloading zone. If move is executed, it moves towards its correct destination with probability P_{move} and in the incorrect direction with probability $1 - P_{move}$ (since the lane is slippery). The other actions have the obvious meanings. The AGV gets a positive reward of 5 when it unloads a job, and gets a penalty of -0.005 whenever it moves with a job. In all other cases, the reward is 0.

Fig. 13(b) shows a DBN for this domain. Typically, a DBN consists of two sets of nodes, where the first set corresponds to the features of a state and the second set to the features of the next state under a given action. Instead, we used a simplified representation in which only the differences between the features in the two states are explicitly shown. The features of the next state can be easily computed by adding these differences to the corresponding features of the current state.

In Fig. 13, ΔJob-on-AGV and ΔAGV-Loc represent the changes to the values of Job-on-AGV and AGV-Loc, respectively. In this domain, ΔJob-on-AGV is either $+1$ for loading, -1 for unloading, and 0 for other actions, and ΔAGV-Loc is either $+1$ for moving right, 0 for staying where it is, and -1 for moving left. Because the load and unload actions are admissible only when Job-on-AGV has the appropriate value, ΔJob-on-AGV is independent of Job-on-AGV or AGV-Loc, given the Action. But Job-on-AGV and Action are both parents of ΔAGV-Loc because the direction of move is dependent on Job-on-AGV as well as Action. Since the action move receives a negative reward only when the AGV has a job, the node Exp-Reward that denotes the expected immediate reward has both Job-on-AGV and Action as its parents.

Learning the CPTs of the dynamic Bayesian network from examples is straightforward because all the features in the network are directly observable and the network structure is known. Consider a node f that has parents $f_1, \ldots, f_k$. The conditional probability $P(f = v \mid f_1 = v_1, \ldots, f_k = v_k)$ is approximated by the fraction of state-action pairs in which f has the value v out of all cases in which the parents have the desired values. For example, the probability of ΔAGV-Loc $= +1$ given Job-on-AGV $= 1$ and Action $=$ move

is the fraction of the cases in which the AGV moved right when it had a job and move was executed.

If n is the number of different AGV locations, the above Bayesian network representation reduces the space requirement of the domain models from $10n - 4$ to 32. An important consequence of this reduction is that learning of domain models can now be much faster. Unfortunately, this is not always easy to see because the learning time is often dominated by the time it takes to learn the value function, and not by the time it takes to learn the domain models. This is true in our Delivery domain. But in domains such as the Slippery-lane, where accurate domain model learning is crucial for performance, Bayesian network-based models can demonstrably expedite policy learning. This will be shown in Section 5.3.1.

5.2. *Value function approximation*

In this section, we describe our value function approximation method that is based on local linear regression.

We chose local linear regression (LLR) as an approximation method for two reasons. First, in an AGV scheduling task such as the Slippery-lane domain of the last section, one of the important features of the state is the location of the AGV. Since the AGV is typically in one of a large set of locations, it is important to be able to generalize the locations to large regions to effectively approximate the h function. Second, in AGV domains the immediate reward is usually independent of the exact location of the AGV given some other features of the state and the action. Under these conditions, the h function is piecewise linear with respect to the AGV location when the other features of the state are constant. In what follows, we assume that the value function is piecewise linear with respect to a small number of such "linear" features and can change arbitrarily with respect to other "nonlinear" features. However, the value function of discounted learning methods like ARTDP is not piecewise linear. Hence using a piecewise linear approximation with discounted learning could introduce larger errors and could require more memory to store the value function.

In linear regression, we fit a linear surface in k dimensions to a set of m data points so that the sum of the squares of the errors of these points with respect to the output surface is minimized [10]. In local linear regression, the data points are chosen in the neighborhood of the point where a prediction is needed. Let us assume that the state is represented by a set of k "linear" features and $n - k$ "nonlinear" features. Our value function approximation is limited to generalizing the values of the k linear features. This is similar to Locally Weighted Regression (LWR) where the nonlinear features are given infinitely large weights [2,29,36]. Instead of representing the h function with a set of piecewise linear functions, we represent the value function using a set of "exemplars", which are a select set of states and their h-values, picked by the learning algorithm. The value function is interpolated for the points between the stored exemplars.

Suppose we need an estimate of $h(p)$, where the state p has values $x_{p1}, \ldots, x_{pn}$ for its n features, where the first k are the linear features. If p is one of the exemplars, its stored value is its estimate. Otherwise, the system first selects the exemplars that

have the same values as p for all its nonlinear features. In other words, these exemplars only differ in the k linear features. Out of these, it picks one nearest exemplar of p in each of the 2^k orthants (subspaces) of the k-dimensional space centered at p. If $k = 1$, for example, the nearest neighbor on each side of p in its first dimension is selected. We found that selecting neighbors this way reduces the potentially large errors due to extrapolating the h-value from states that are far off from p but are close to each other. The distance between states that differ in their linear features is measured by the Euclidean distance. After selecting the 2^k neighbors, the system uses linear regression on the values of the first k features of these exemplars to find a least squares fit and uses it to predict its h-value. If the predicted value is greater than the maximum of the values of all its selected neighbors, or is less than the minimum of their values, it is set to the maximum or minimum of their values, respectively. This step is useful in reducing the errors due to large differences in the distances of different neighbors from p.

If p does not have neighbors in all the 2^k orthants that share its values for all their nonlinear features, then the number of dimensions k is reduced by 1, and the nearest neighbors in 2^{k-1} orthants are selected such that the above condition holds. This is continued until $k = 1$. If this also fails to find such neighbors, then the h-value is estimated to be 0.

At any time, an exemplar is stored if its h-value cannot be estimated within a given tolerance from the currently stored exemplars. Since the h-values of adjacent states in the state space differ by ρ when there is no immediate reward, the tolerance is normalized by multiplying a constant parameter ε with ρ. An exemplar is also stored if its updated h-value is greater or less than the h-values of all its selected nearest neighbors. Whenever a new exemplar (i, v) is stored, the system checks to see if any of its nearest neighbors (selected as described above), say j, can be safely deleted from the exemplar set, because it may now be possible to approximate the stored $h(j)$-value from j's nearest neighbors, which may include i. Without this heuristic, a large number of exemplars will be stored in the beginning, and they are never deleted afterwards.

We call the version of H-learning that learns Bayesian network action models and approximates the value function using local linear regression, "LBH-learning".

Fig. 14 shows the algorithm of LBH-learning. It initializes the exemplar set to empty, ρ to a value higher than the expected optimum gain, and α to a low value. The explicitly stored h-values of the exemplars are denoted by $h(.)$, and the values estimated by LLR are denoted by $\hat{h}(.)$. The immediate rewards $r_i(u)$ and the transition probabilities $p_{i,j}(u)$ can be inferred from Bayesian networks using the standard Bayesian network inference algorithms. The parameters of the Bayesian networks are updated incrementally by Update-BN-model as described in Section 5.1. The exemplars are updated and used in the prediction of h-values of other states as described in the previous paragraphs.

Our algorithm is similar to the edited nearest neighbor algorithms, which collect exemplars that improve their predictive accuracy and prune the unnecessary ones to keep the size of the representation small [1, 13, 14]. One problem with the edited nearest neighbor approaches is that they are highly sensitive to noise, since they tend to

1. Take an exploratory action a or a greedy action a that maximizes $r_i(a) + \sum_{j=1}^{n} p_{i,j}(a)\hat{h}(j)$. Let l be the resulting state, and r_{imm} be the immediate reward received.
2. Update-BN-model (i, a, l, r_{imm})
3. If a is a greedy action, then
   (a) $\rho \leftarrow (1-\alpha)\rho + \alpha(r_i(a) + \hat{h}(l) - \hat{h}(i))$
   (b) $\alpha \leftarrow \frac{\alpha}{\alpha+1}$
4. $v \leftarrow \max_u \{r_i(u) + (\sum_{j=1}^{n} p_{i,j}(u)\hat{h}(j))\} - \rho$
5. If $(i, h(i)) \in$ Exemplars, delete it. Let Neighbors be the nearest neighbors of i in the 2^k orthants surrounding i that differ from it only in the values of k linear features.
6. If $|v - \hat{h}(i)| > \varepsilon\rho$ or $v > \max_{j \in \text{Neighbors}} h(j)$ or $v < \min_{j \in \text{Neighbors}} h(j)$
   (a) Add (i, v) to Exemplars.
   (b) For any $j \in$ Neighbors, if $|h(j) - \hat{h}(j)| \leqslant \varepsilon\rho$ then delete $(j, h(j))$ from Exemplars. ($\hat{h}(j)$ is computed after temporarily removing $(j, h(j))$ from Exemplars.)
7. $i \leftarrow l$

Fig. 14. LBH-learning, which uses Bayesian networks for representing the action models and local linear regression for approximating the value function. Steps 1–7 are executed when the agent is in state i.

store all the noise points, which cannot be interpolated from the remaining points [1]. Our algorithm does not seem to suffer from this problem, since the target value function is in fact piecewise linear and has no noise. Even though the intermediate examples that the learner sees are indeed noisy, the value function still appears to be piecewise linear (with a small number of "pieces") even in the intermediate stages of learning. This is because the value function is updated by backing up the values one step at a time from the adjacent states, which keeps it locally consistent and hence locally linear in almost all places.[4]

Like H-learning, LBH-learning can be extended to "Auto-exploratory LBH-learning" (ALBH-learning) by initializing ρ to a high value and explicitly storing R-values instead of h-values. Thus ALBH-learning uses Bayesian networks and local linear regression for approximation and converges to the gain-optimal policy by taking only greedy actions. It maintains a separate set of exemplars for each action a to store the tuples $\langle i, R(i, a)\rangle$. In the next section, we evaluate different versions of H-learning including LBH-learning and ALBH-learning in several domains.

5.3. Experimental results

In this section, we show experimental results in three different AGV-scheduling domains. We compare the performance of six versions of H-learning. Each version is named by its extensions: A for using auto-exploration, B for learning Bayesian network

[4] There are, in fact, two kinds of locality: locality in the state space and locality in the Cartesian space inhabited by the AGV. The fact that these two notions coincide in our domain is an important premise behind the above argument.

action models, and L for approximating the value function using local linear regression.

We also compare the performance of ARTDP with Bayesian network action models and local linear regression (LBARTDP), and without these two extensions (ARTDP). Since LBARTDP does not have the parameter ρ, its tolerance was normalized to be $\varepsilon\beta_i$, where β_i is the smallest local slope of the value function in state i among all its linear dimensions. This gave better performance than using a constant tolerance.

Because H-learning with a high value of ρ gives good performance, all algorithms based on H-learning start with a high value of ρ. In all these experiments, the algorithms that do not have the auto-exploratory component take random actions with $\eta = 0.1$ probability while the auto-exploratory algorithms always take greedy actions. The experiments demonstrate the synergy between Bayesian network models and local linear regression, the scalability of the learning methods in both space and time, and their applicability to domains with multiple linear features.

5.3.1. Improving the performance of H-learning

The goal of the first experiment is to demonstrate the synergy between local linear regression and the Bayesian network learning in scaling H-learning.

We use the Slippery-lane domain shown in Fig. 13(a). The dynamic Bayesian networks in Fig. 13(b) are used for representing the domain models. The number of locations for the AGV was set to 30. To make the model learning significantly important, P_{move} was set to 0.6, i.e., with 60% probability, the AGV moves to the unloading zone if it has a job and to the loading zone if it has no job.

The parameters of each method were tuned by trial and error. The ρ-values were initialized to 0.1, 0.05, 0.02, 0.01, 0.01, and 0.01 respectively for ALBH-learning, LBH-learning, LH-learning, BH-learning, AH-learning, and H-learning. The α-values were initialized to 0.001, 0.005, 0.0001, 0.001, 0.001, and 0.001 respectively. For LBH-learning, LH-learning and LBARTDP, ε was set to 1, 1, and 2 respectively. The discount factor for LBARTDP was set to 0.95. The experiments were repeated for 30 trials for each method, starting from a random initial state. In Fig. 15, we plotted the off-line average reward over 30 trials, each estimate being based on 3 runs of 100K steps from 3 random start states. We chose off-line estimation because the convergence is too fast to reliably measure the on-line average reward in this domain.

Since P_{move} is close to 0.5, model learning is very important in this domain. Thus, BH-learning, with its parameterized domain models converged more quickly than H-learning which stored its models as tables. LH-learning also converged faster than H-learning, but due to a different reason, namely value function approximation. Most interesting was the performance of LBH-learning which was clearly superior to both BH-learning and LH-learning, thus demonstrating the synergy between the two kinds of approximations used. Also ALBH-learning converged faster than both BAH-learning and AH-learning, and both BAH-learning and AH-learning converged faster than H-learning. Since AH-learning explored the domain effectively and learned models faster, BAH-learning converged only slightly faster than AH-learning. Because ALBH-learning updates one R-value of state-action pair at every step while LBH-learning updates

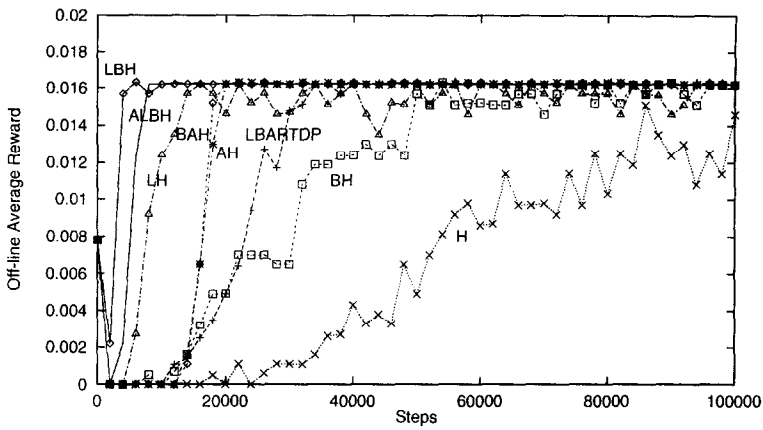

Fig. 15. Off-line average reward of $ALBH^{0.1,0.001}$, $BAH^{0.02,0.001}$, and $AH^{0.01,0.001}$ without random exploration and $LBH^{0.1,0.002}$, $BH^{0.01,0.001}$, $LH^{0.1,0.0005}$, $H^{0.01,0.001}$, and LBARTDP with random exploration with $\eta = 0.1$, averaged over 30 trials. In each trial, evaluation is over 3 off-line runs of 100 K steps each from 3 random start states at each point.

one h-value of state at every step, ALBH-learning converged a little slower than LBH-learning. LBARTDP was slower than LBH-learning, ALBH-learning, BAH-learning, and AH-learning, although it was faster than BH-learning and H-learning.

Fig. 16(a) shows the average exemplar size of all methods that do not use auto-exploration. In the end, LBH-learning stores only 4 states that correspond to the end locations of the slippery-lane, while LBARTDP stores 6 or 7 states because the value function of ARTDP is not piecewise linear. LH-learning also stores 6 or 7 states because its value function cannot be very smooth without the Bayesian network models. BH-learning and H-learning, on the other hand, store values for all 60 states. Fig. 16(b) shows the average exemplar size of all auto-exploratory methods. These methods store more exemplars than the methods in Fig. 16(a) because they have to approximate the R-value for each state and each action. However, in the end, ALBH-learning stores only 17 exemplars of R-values while BAH-learning and AH-learning store all 122 exemplars of R-values.

LBH-learning stores 6.7% of all exemplars of h-values and ALBH-learning stores 13.6% of all exemplars of R-values. Because the auto-exploratory methods take only greedy actions, the suboptimal actions may not be taken frequently enough to make the value functions for those actions smooth. Therefore, even though ALBH-learning uses the same approximation methods as LBH-learning, ALBH-learning stores a greater proportion of the total exemplars than LBH-learning. All H-learning methods using local linear regression store more exemplars in the beginning than in the end because the value function may not be piecewise linear when it is not fully converged, and hence they need more exemplars to store it accurately.

These results suggest that local linear regression and Bayesian network models improve both the time and space requirements of H-learning, and make it converge much faster than its discounted counterpart.

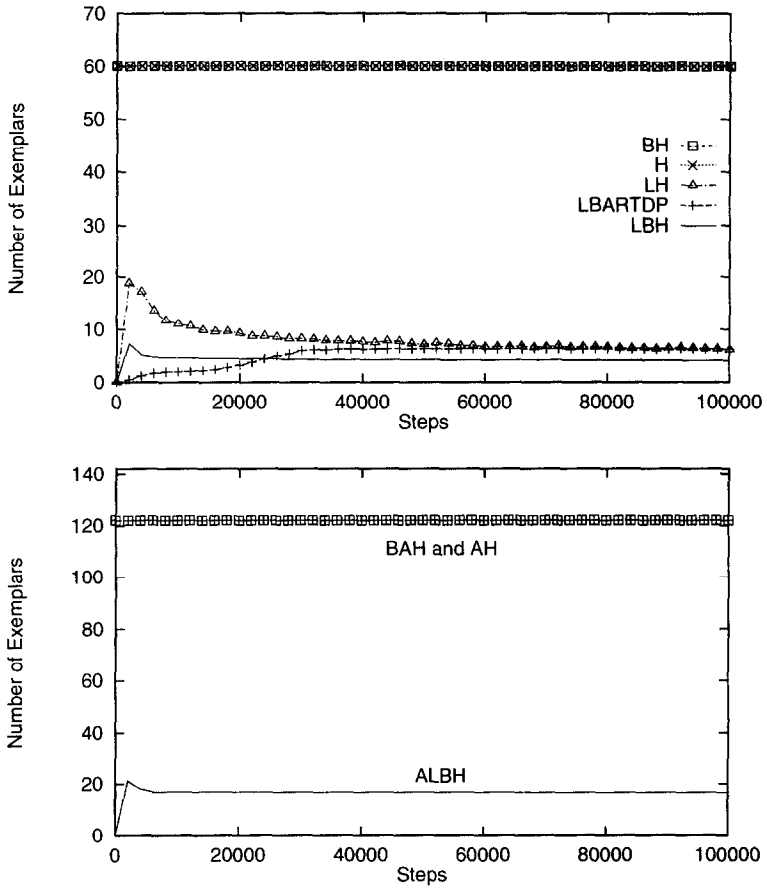

Fig. 16. Number of exemplars for (a) $BH^{0.01,0.001}$, $H^{0.01,0.001}$, $LBH^{0.05,0.005}$, $LH^{0.02,0.0001}$, and LBARTDP (top), and (b) $AH^{0.01,0.001}$, $BAH^{0.02,0.001}$, and $ALBH^{0.1,0.001}$ (bottom). All methods used random exploration with $\eta = 0.1$. Each point is the mean of 30 trials.

5.3.2. *Scaling of LBH-learning with domain size*

The goal of the experiment of this section is to demonstrate the scaling of LBH-learning and ALBH-learning by varying the size of the domain. We use the "Loop domain" of Fig. 17(a) to do this.

There are one AGV, two job generators 1 and 2, and two conveyor belts 1 and 2. Each generator generates jobs for each destination belt with 0.5 probability. A state is described by 5 features—Lane, AGV-Loc, Job-on-AGV, Job-at-Gen1, and Job-at-Gen2—with obvious meanings. The variable Lane takes values from 1 to 4 and denotes the lane number of the AGV's location as shown in Fig. 17(a). The total number of possible locations of the AGV is denoted by n. There are $n/5$ locations in each of the short lanes and $2n/5$ locations in lane 1. AGV-Loc takes values from $1, \ldots, 2n/5$ for lane 1 but from $1, \ldots, n/5$ for the other three lanes. Job-on-

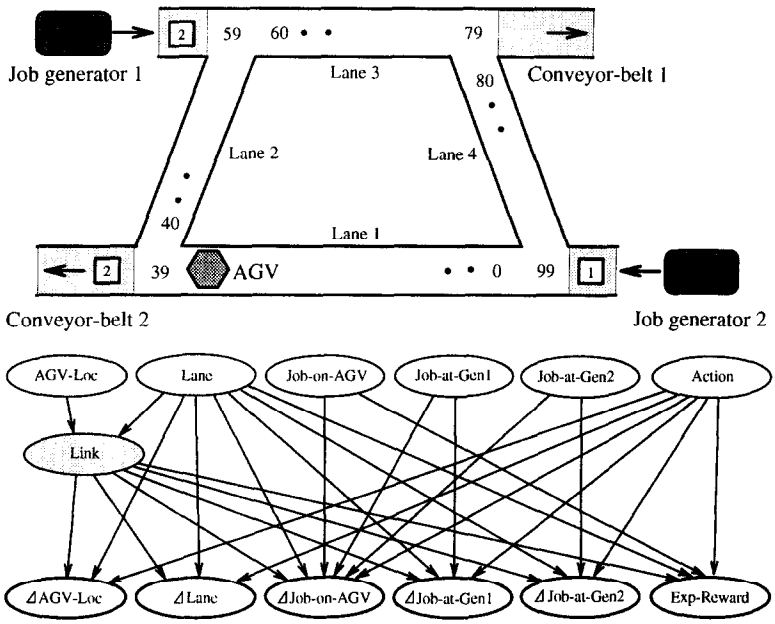

Fig. 17. (a) The Loop domain (top) and (b) its dynamic Bayesian network (bottom).

AGV is 0 if the AGV has no job and indicates the destination of the job (1 or 2) if it has a job. Job-at-Gen1 and Job-at-Gen2 are 1 or 2 depending on the destination of the job waiting at the generators. Therefore, the size of the state space is $n \times 3 \times 2 \times 2 = 12n$. The AGV has 4 actions move-forward, move-backward, load and, unload. The AGV can always take the move-forward action or the move-backward action. If the AGV does not have a job it can take the action load at the loading zones, and if it has a job it can take the action unload at the unloading zones. To make the optimal average rewards the same for all n, the immediate reward for delivery is made proportional to n. If the AGV delivers a job to its correct belt, it receives a reward of $+0.1n$. If it delivers it to the wrong belt, it gets a smaller reward of $+0.02n$. The goal of the AGV is to move jobs from job generators to proper conveyor-belts. Whenever the AGV loads a job from job generator, a new job is generated.

Fig. 17(b) shows the dynamic Bayesian network for this domain. One new feature, Link, abstracts the AGV-Loc feature. Link takes 3 values: *end1* and *end2* for the two end locations of the lane, and *middle* for the other locations between the two ends. This feature distinguishes the end locations of each lane from the rest, which is useful to succinctly model AGV's motion in the loop. Since ΔAGV-Loc is now independent of AGV-Loc, given Link, Lane and Action, this representation reduces the space requirements to store the domain models from $48n + 52$ to 1056.

We used random exploration with $\eta = 0.1$ in this experiment for H-learning and LBH-learning. AH-learning and ALBH-learning use auto-exploration. We set ε to 1 and

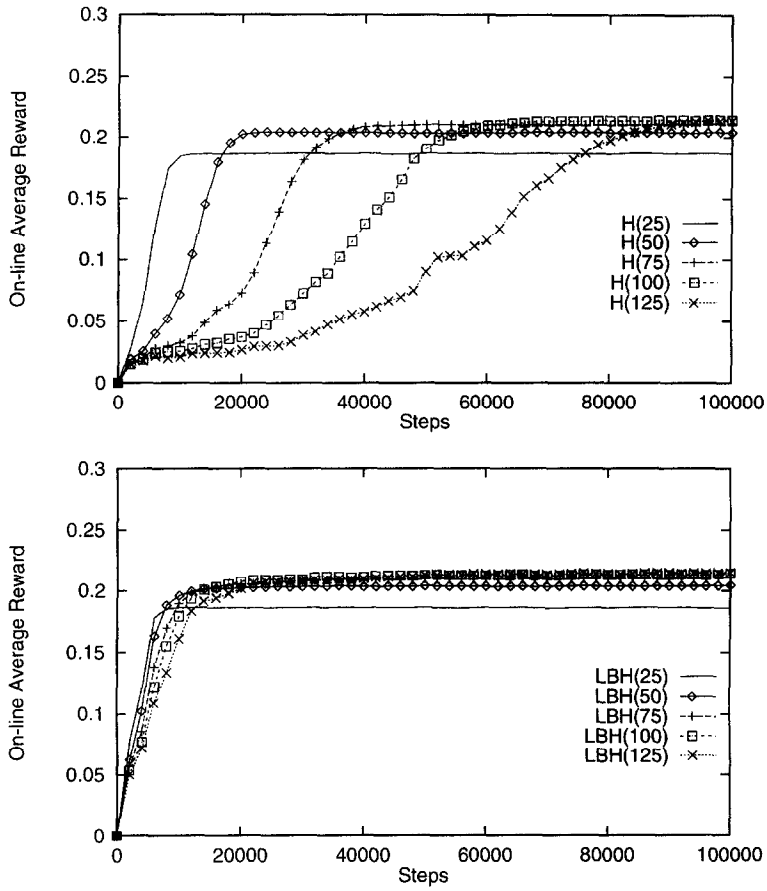

Fig. 18. On-line average rewards per step for H and LBH in the Loop domain estimated over the last 5000 steps. Each point is the mean of 30 trials. (a) $H^{1,0.02}(25)$, $H^{1,0.005}(50)$, $H^{1,0.002}(75)$, $H^{1.5,0.003}(100)$, $H^{1.5,0.002}(125)$ (top), and (b) $LBH^{1,0.02}(25)$, $LBH^{1.3,0.007}(50)$, $LBH^{1.5,0.004}(75)$, $LBH^{1.8,0.004}(100)$, and $LBH^{2,0.003}(125)$ (bottom), where the superscripts are ρ_0 and α_0, and the number in the parentheses is the total number of AGV locations. ε for ALBH-learning is set to 1.

varied n from 25 to 125 in steps of 25. The parameters of each algorithm for each value of n are tuned by trial and error and are shown in the captions of Figs. 18 and 19. The on-line average rewards of H-learning and LBH-learning are shown in Fig. 18 and those of AH-learning and ALBH-learning are shown in Fig. 19. The values shown are estimated on-line over the last 5000 steps and are averaged over 30 trials.

As we can see, the convergence speeds of LBH-learning and ALBH-learning are consistently better than those of H-learning and AH-learning, and the difference increases with n. The convergence takes longer for larger values of n, because the AGV has to travel over longer distances to get new information. However, in LBH-learning and ALBH-learning, the number of steps for convergence grows much more slowly than in

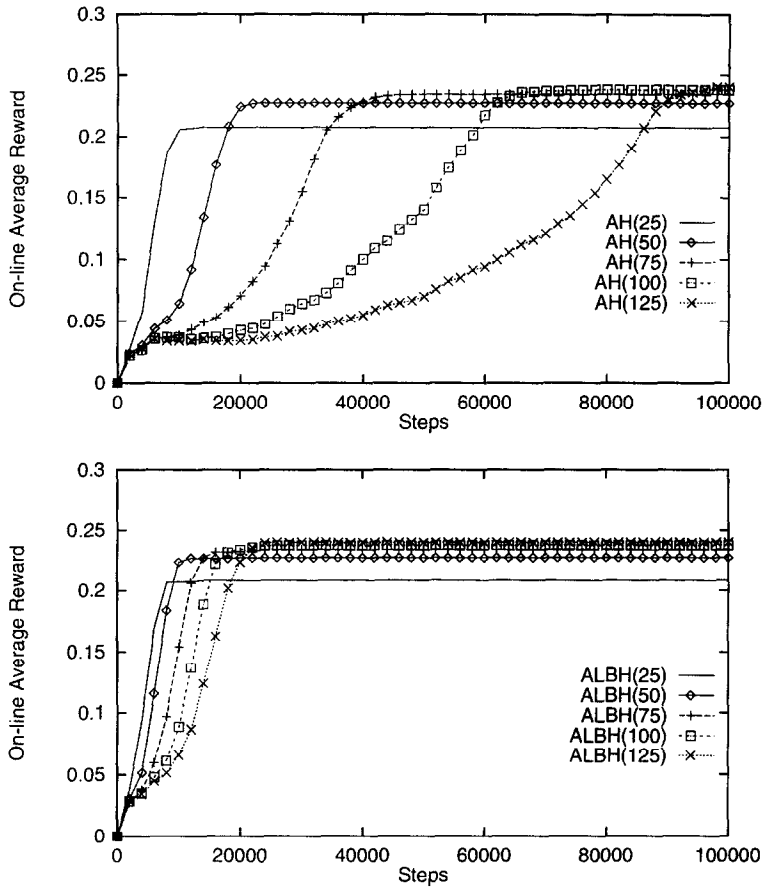

Fig. 19. On-line average rewards per step for AH and ALBH in the Loop domain estimated over the last 5000 steps. Each point is the mean of 30 trials. (a) $AH^{1.0.002}(25)$, $AH^{1.0.0008}(50)$, $AH^{1.5.0.0004}(75)$, $AH^{1.5.0.0002}(100)$, $AH^{1.2.0.0001}(125)$ (top), and (b) $ALBH^{1.0.003}(25)$, $ALBH^{1.5.0.003}(50)$, $ALBH^{1.7.0.002}(75)$, $ALBH^{2.0.002}(100)$, and $ALBH^{2.0.002}(125)$ (bottom), where the superscripts are ρ_0 and α_0, and the number in the parentheses is the total number of AGV locations. ϵ for ALBH-learning is set to 1.

H and AH. Also the on-line average rewards of AH-learning and ALBH-learning are higher than those of H-learning and LBH-learning respectively for each value of n. This difference is attributable to two factors. First, the rates of exploration of H-learning and LBH-learning are not decayed in this experiment, whereas the auto-exploratory methods have a built-in decay mechanism. Perhaps more importantly, as the experiments in Section 4.2 illustrate, auto-exploratory methods are more effective in exploring only potentially useful parts of the state space.

The numbers of the stored exemplars of LBH-learning and ALBH-learning, averaged over 30 trials, are shown in Fig. 20 in comparison to the exemplars stored by H-learning and AH-learning. The bigger the value of n, the larger the number of exemplars stored

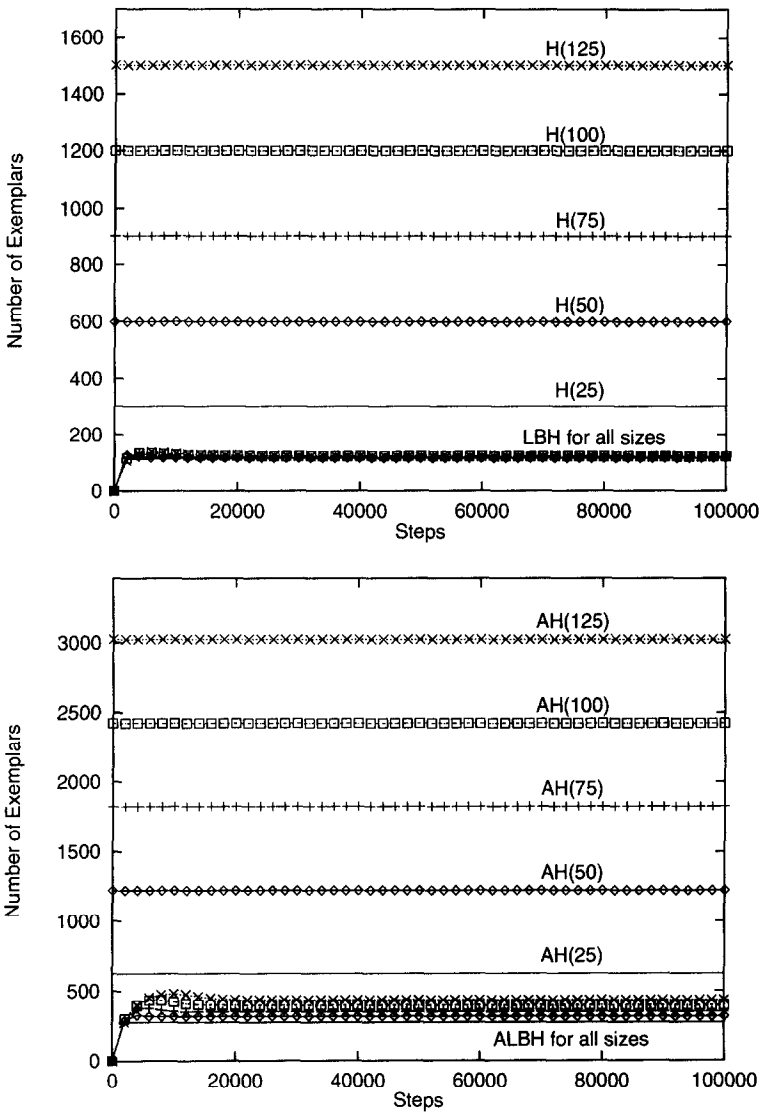

Fig. 20. Number of exemplars of (a) H-learning and LBH-learning (top) with random exploration with $\eta = 0.1$, and (b) AH-learning and ALBH-learning (bottom) with only auto-exploration in the Loop domain. The results are mean averages over 30 trials.

by H-learning and AH-learning. For LBH-learning, the absolute number of exemplars almost remains constant with increasing n. ALBH-learning performs like LBH-learning but the absolute number of exemplars increases slightly more than LBH-learning by increasing n. This is because its R-value function is less smooth than the h function, especially for suboptimal actions.

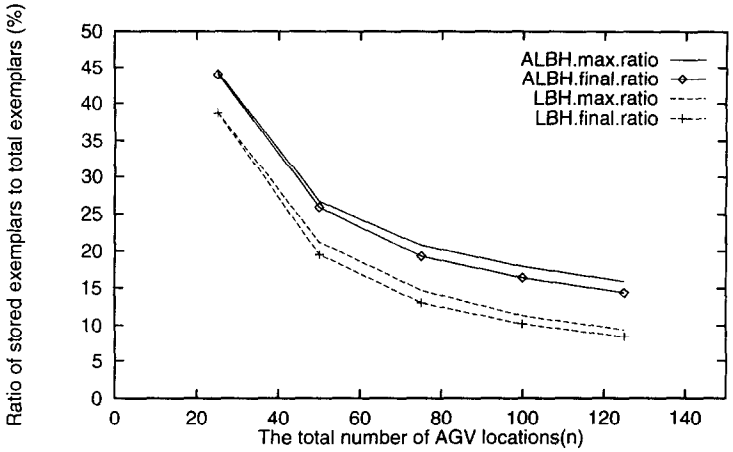

Fig. 21. The maximum and final ratios of stored exemplars to all possible exemplars for LBH-learning and ALBH-learning as a function of the domain size n in the Loop domain. The results are mean averages over 30 trials.

Fig. 21 shows the percentage of the number of stored exemplars out of the total exemplars as a function of the total number of possible locations (n) of the AGV in this domain. LBH.max.ratio and ALBH.max.ratio indicate the maximum values of this ratio for LBH-learning and ALBH-learning during learning (usually at the beginning) and LBH.final.ratio and ALBH.final.ratio indicate the values of this ratio for LBH-learning and ALBH-learning after the last step. The ratio is maximum at 38.8% for LBH-learning and 44.3% for ALBH-learning when the total number of AGV locations $n = 25$, and gradually reduces to 9.4% for LBH-learning and to 15.9% for ALBH-learning when $n = 125$. The final ratios are lower than the maximum ratios because the value function at the end is a lot smoother than it is in the beginning. Because ALBH-learning always takes greedy actions, its R-values for suboptimal actions are not fully converged. Hence it stores more exemplars than LBH-learning. Fig. 21 clearly shows that LBH-learning and ALBH-learning store far fewer exemplars than H-learning and AH-learning as n increases.

The value functions for H-learning, LBH-learning, AH-learning, and ALBH-learning at the end of training are shown in Fig. 22 in comparison to the optimal value function. The value functions for H-learning and LBH-learning are smoother than those for AH-learning and ALBH-learning because H-learning and LBH-learning take random actions with 0.1 probability and thus explore the state space more evenly, while AH-learning and ALBH-learning take only greedy actions and thus do not thoroughly explore the suboptimal regions of the state space. A learning method based on H-learning usually has a smooth piecewise linear value function when using random exploration, which forces the agent to visit all states more or less uniformly. Due to local linear regression, LBH-learning's value function is smoother than H-learning's value function and ALBH-learning's value function is smoother than AH-learning's value function. Thus, LBH-learning's value function is the closest to the optimal value function, shown in solid lines in Fig. 22.

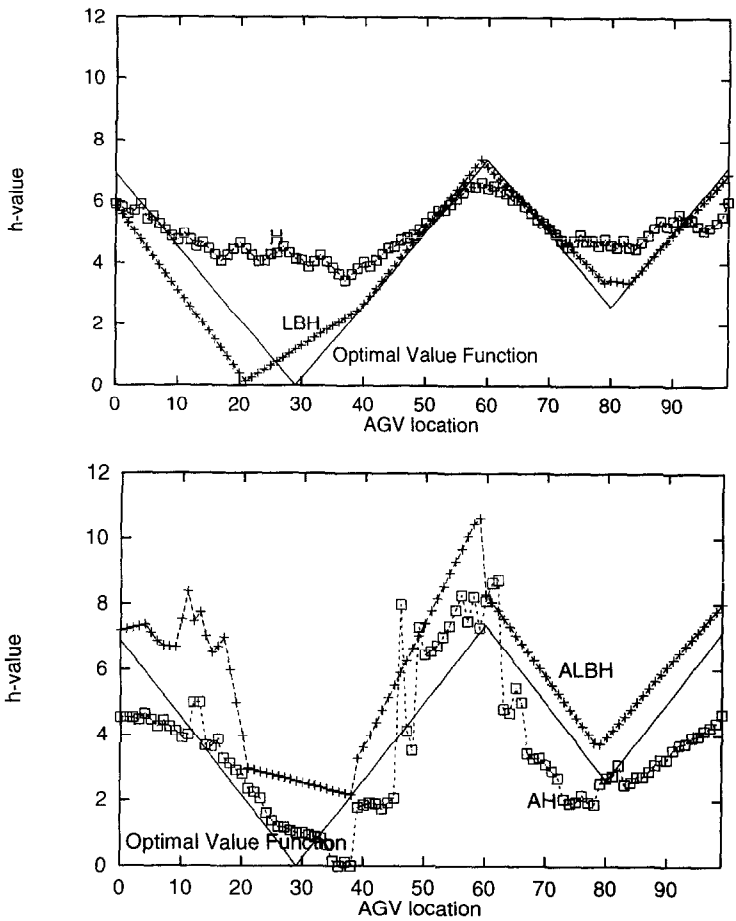

Fig. 22. The h-values as a function of AGV location for (a) H and LBH (top) and (b) AH and ALBH (bottom), when AGV does not have a job, generator 1 has job 2 and generator 2 has job 1, and $n = 100$. The AGV location is as shown in Fig. 17(a). The optimal value function is shown in solid lines in the two plots.

The results in this domain show that LBH-learning and ALBH-learning scale better than H-learning and AH-learning with the domain size, both in terms of learning speed and memory use.

5.3.3. *Scaling to multiple linear features*

In this section, we demonstrate LBH-learning and ALBH-learning in the "Grid" domain shown in Fig. 23(a), which has two linear dimensions.

At one corner of the 15×15 grid, there is a job generator and at the opposite corner, there is a destination conveyor-belt. The AGV can take any action among four admissible actions—move-north, move-south, move-east, and move-west—if there is no obstacle or wall in the location the AGV wants to move to. The dark squares in the middle represent the obstacles. The AGV also has the load or the unload action available at the

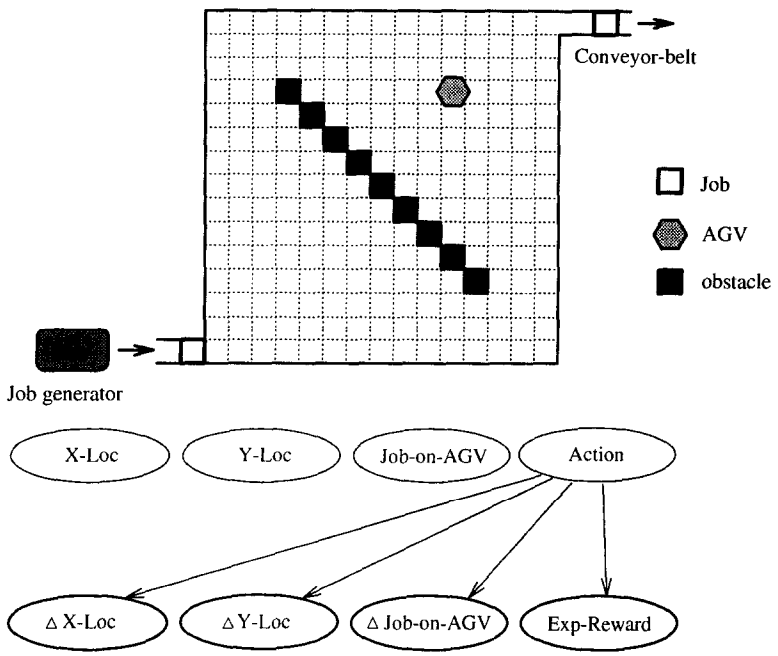

Fig. 23. (a) The Grid domain (top) and (b) its dynamic Bayesian network (bottom).

appropriate corner. It receives a reward of 58 when it delivers a job, so that the optimal average reward per step is 1. Fig. 23(b) shows the dynamic Bayesian network for the Grid domain, which is very compact since the effect of any available action depends only on the Action variable. In fact, it reduces the space requirements for storing the domain model from 3076 to 24.

Each point in Fig. 24 is the on-line average reward for 30 trials calculated over the last 5000 steps. It shows the effect of setting ε to 0.4, 0.7, 1.0, and 1.3 on the average reward of LBH-learning with random exploration with $\eta = 0.1$, and on the average reward of ALBH-learning. With $\varepsilon = 0.4$, 0.7, and 1.0, LBH-learning finds the gain-optimal policy much faster than H-learning does. With $\varepsilon = 1.3$, its speed decreases, because too high a value of ε sometimes makes it converge to a suboptimal policy. Similarly, ALBH-learning with $\varepsilon = 0.4$, 0.7, and 1.0 finds the gain-optimal policy much faster than AH-learning does. However ALBH-learning with $\varepsilon = 1.3$ converges to a suboptimal policy, as ε is too high and it does not take any random exploratory actions. Thus its average reward remains zero.

Fig. 25 shows the number of exemplars stored by LBH-learning and ALBH -learning. LBH-learning and ALBH-learning usually store fewer exemplars with higher values of ε, except when ε is too high, i.e., 1.3 in this case. When ε is too high, learning methods converge to a suboptimal policy and do not sufficiently explore the state space. The value functions of LBH-learning and ALBH-learning are not smooth without sufficient exploration of the state space, which makes them store a large number of exemplars.

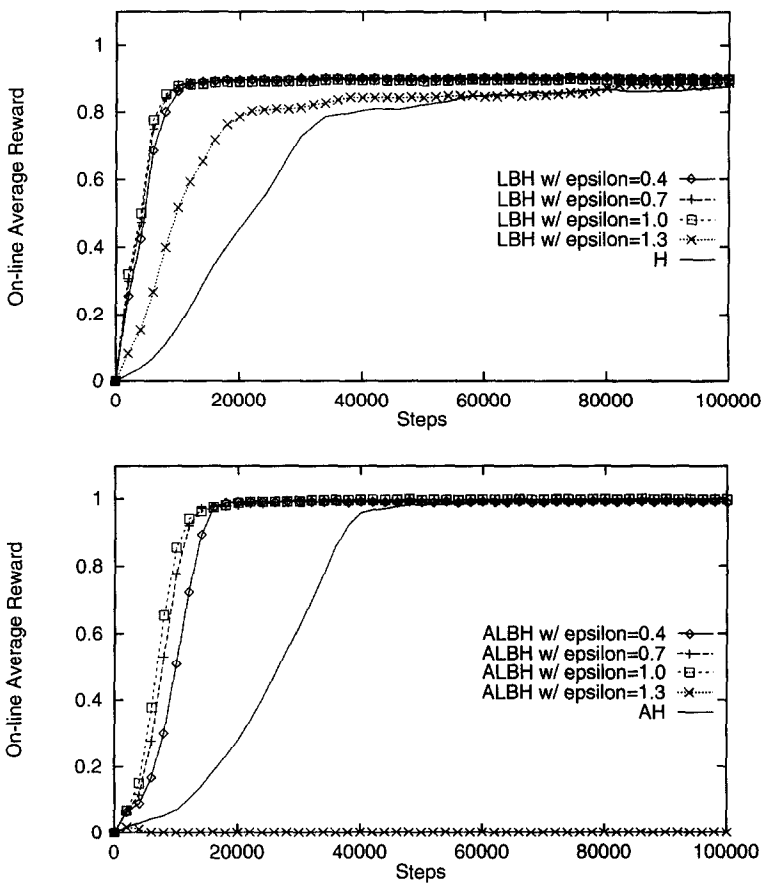

Fig. 24. On-line average rewards per step for (a) $H^{3.0.003}$ and $LBH^{3.0.002}$ (top) with random exploration with $\eta = 0.1$, and (b) $AH^{8.0.0003}$ and $ALBH^{5.0.0007}$ (bottom) without exploration in the Grid domain. Each point is the mean of 30 trials over the last 5,000 steps.

The final value functions of H-learning, AH-learning, LBH-learning, and ALBH-learning are shown in Fig. 26 in comparison to the optimal value function. For this domain, the true optimal value function of H-learning is piecewise linear with respect to AGV's location. LBH-learning and ALBH-learning approximate this value function for multiple linear features successfully using local linear regression.

Since the obstacles in the middle of the grid are not states, they do not have h-values. Some arbitrary values lower than the h-values of states surrounding them are assigned to their locations in the plots of Fig. 26. Like the experimental results in the Loop domain (Fig. 22), the value functions for H and LBH (in the left half of Fig. 26) are smoother than the corresponding auto-exploratory versions (in the right half). The value functions for LBH and ALBH learning are smoother than those without approximation (H and AH, respectively).

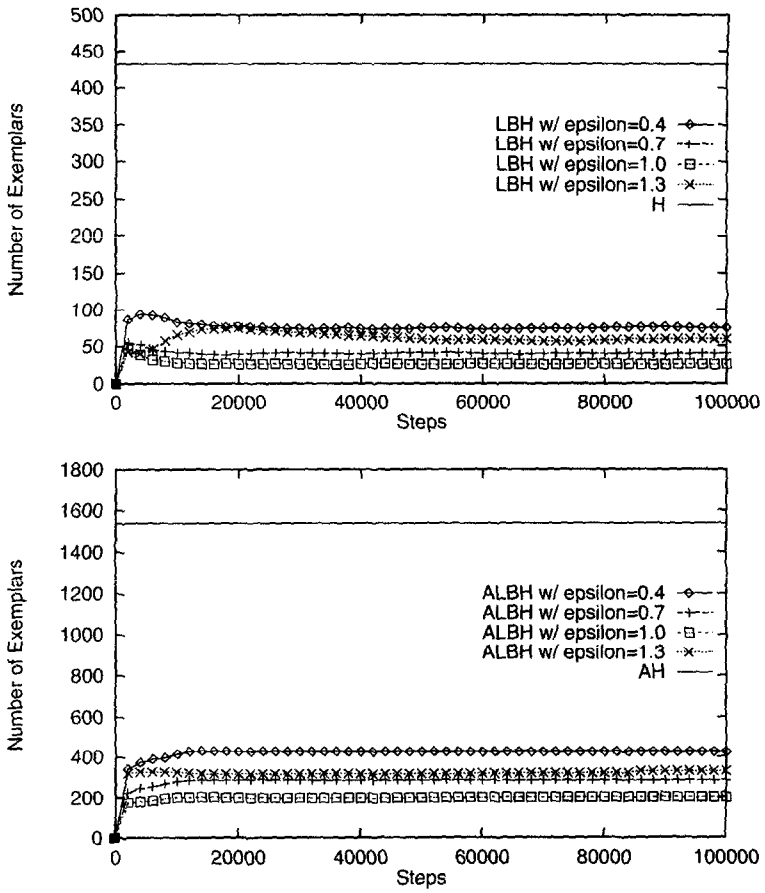

Fig. 25. Average number of exemplars of 30 trials of (a) $H^{3,0.003}$ and $LBH^{3,0.002}$ (top) with random exploration ($\eta = 0.1$) and (b) $AH^{8,0.0003}$ and $ALBH^{5,0.0007}$ (bottom) without exploration in the Grid domain.

The results of this section demonstrate that local linear regression and Bayesian network model learning extend to two-dimensional spaces, and significantly reduce the learning time and memory requirements.

6. Discussion and future work

The basic premise of this paper is that many real-world domains demand optimizing average reward per time step, while most work in Reinforcement Learning is focused on optimizing discounted total reward. Because discounting encourages the learner to sacrifice long-term benefits for short-term gains, using discounted RL in these domains could lead to suboptimal policies. We presented a variety of algorithms based on H-

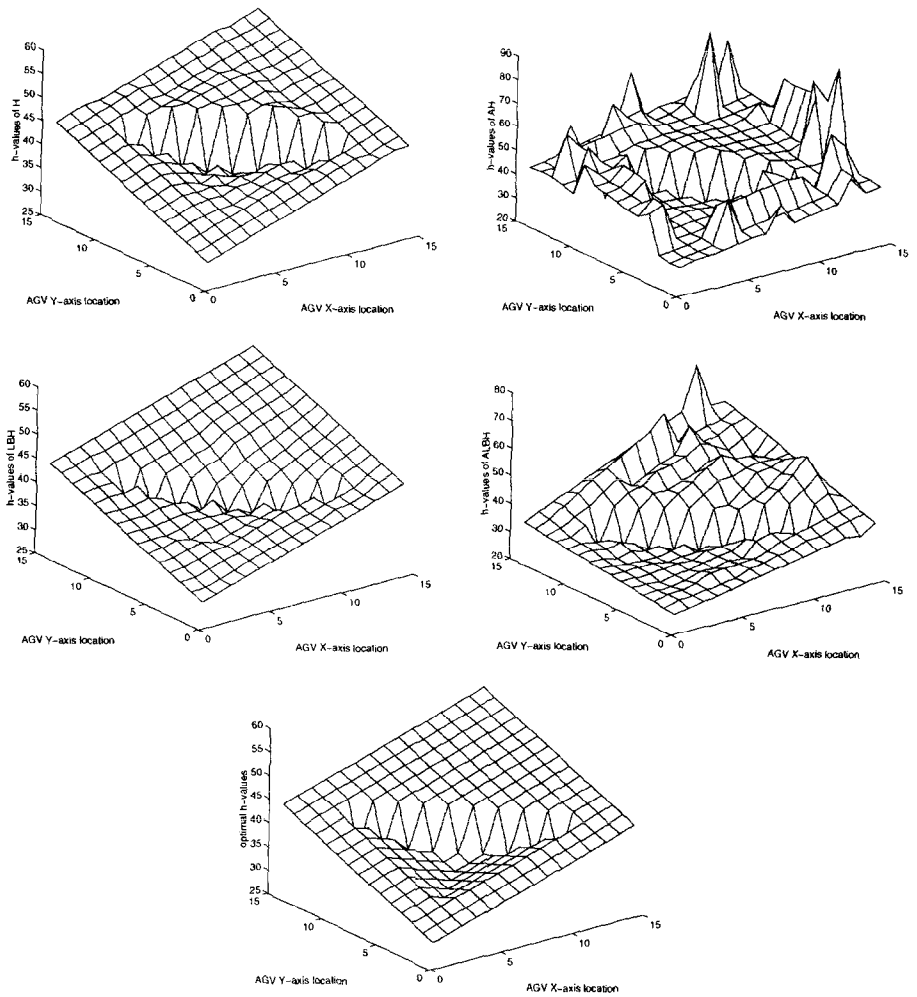

Fig. 26. The value functions of H-learning (top left), AH-learning (top right) LBH-learning (middle left), and ALBH-learning (middle right), in comparison to the optimal value function (bottom) in the Grid domain. H-learning and LBH-learning use random exploration with $\eta = 0.1$. ε for LBH-learning and ALBH-learning is set to 1.

learning, a model-based method designed to optimize the gain or average reward per time step, and demonstrated their usefulness in AGV scheduling tasks. Earlier presentations of parts of this work include [41], [32], and [42].

Fig. 27 shows the family of algorithms obtained by adding auto-exploration, Bayesian network model learning, and local linear regression to H-learning. We can choose any combination of these three extensions, depending on the domain, our needs, and the resources and the prior knowledge available. Based on our experiments, we can make the following recommendations:

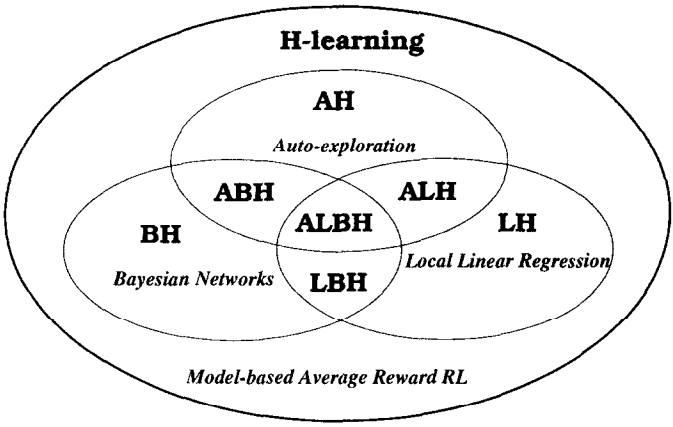

Fig. 27. The family of algorithms based on H-learning.

- When we know an upper bound $\rho_{\max}$ on the gain,
  - use auto-exploration with ρ initialized to something higher than $\rho_{\max}$.
- When we cannot afford tuning the exploration parameters,
  - use H-learning with a small $\rho > 0$ and $\alpha = 1$ with some exploration strategy.
- When we have only a limited amount of space and time,
  - use local linear regression.
- When the structures of the Bayesian network action models are available,
  - use them to learn parameters for the Bayesian network action models.

There is an extensive body of literature on average-reward optimization using dynamic programming approaches [6, 15, 34]. Mahadevan gives a useful survey of this literature from Reinforcement Learning point of view [24]. Schwartz and Singh present model-free RL algorithms for average-reward optimization [37, 38]. There are at least two average-reward reinforcement learning methods that have been proved to converge under suitable conditions [4, 16]. Bertsekas's algorithm is based on converting the Average-reward RL problem into a stochastic shortest path algorithm with slowly changing edge costs [4, 6]. The edge costs are essentially the negated average-adjusted immediate rewards, i.e., $\rho - R_i(u)$, where ρ is the gain of the current greedy policy and $R_i(u)$ is the immediate reward of executing action u in state i. Hence, it uses a recurrence relation that is equivalent to Eq. (7) in Section 3 for updating the h values. ρ is estimated by on-line averaging of the h value of a reference state rather than by on-line averaging of adjusted immediate rewards, as in H-learning. The basic H-learning algorithm is very similar to the Algorithm B of Jalali and Ferguson [16]. The main difference is due to exploration, which is ignored by Jalali and Ferguson.

In this paper, we have been mainly concerned with average-reward optimality or gain-optimality. Bias-optimality, or Schwartz's T-optimality, is a more refined notion than gain-optimality [34, 37]. It seeks to find a policy that maximizes the expected total reward obtained before entering a recurrent state, while also being gain-optimal. All gain-optimal policies are not bias-optimal. H-learning and R-learning can find the

bias-optimal policies for unichain MDPs only if all gain-optimal policies give rise to the same recurrent set of states, and all states, including the non-recurrent states, are visited infinitely often using some exploration strategy. To find the bias-optimal policies for more general unichains, it is necessary to select bias-optimal actions from among the gain-optimal ones in every state using more refined criteria. Mahadevan extends both H-learning and R-learning to find the bias-optimal policies for general unichains [23,25]. His method is based on solving a set of nested recurrence relations for two values h and W for each state, since the h values alone are not sufficient to determine a bias-optimal policy. Bias optimal policies appear to have a significant advantage in some domains such as the admission control queuing systems studied in operations research literature [23,25].

Auto-exploratory H-learning belongs to the set of exploration techniques that can be classified as "optimism under uncertainty". The general idea here is to initialize the value function and the reward functions so that a state (or state-action pair) that is not sufficiently explored appears better than a well-explored state, even if the latter is known to have a relatively high utility. Koenig and Simmons achieve this effect simply by zero-initializing the value function and by giving a negative penalty for each action [20]. In deterministic domains, or when the updates are done using the minimax scheme, this ensures that the Q-values of state-action pairs are never less than their true optimal values. In analogy to the A* algorithms, this is called the "admissibility" condition. The more frequently an action is executed, the more closely its Q-value approaches its real value. Hence choosing to execute an action with the highest Q-value will have one of two effects: in one case, its Q-value may forever remain higher than the Q-values of other actions in the same state. Since the Q-values of other actions are non-underestimating, this implies that the executed action is actually the best. In the other case, its Q-value would eventually decrease to a value below those of the other available actions in the same state. This gives the algorithm a chance to execute the other potentially best actions, thus encouraging exploration.

Kaelbling's Interval Estimation (IE) method is based on a more sophisticated version of the same idea, and is applicable to stochastic domains [17]. It maintains a confidence interval of the value function for each state, and picks actions that maximize the upper-bounds of their confidence intervals. If the true value of a state (or state-action pair) is in the confidence interval with a high probability, we can say that the upper bounds of the confidence intervals satisfy the above admissibility property with a high probability. Hence, picking actions using this upper bound will have the desired auto-exploratory effect.

In AH-learning, the same effect is achieved by initializing ρ to a high value and the R values to 0s. Since ρ is subtracted in the right-hand side of the update equation of AH-learning, it is equivalent to giving a high penalty for each action as in Koenig and Simmons method [20]. As the system converges to the optimal policy, ρ converges to the optimal gain, and the system stops exploring states not in the optimal loop. Theoretically characterizing the conditions of convergence of AH-learning is an important open problem. The idea of Auto-exploratory learning can be adapted to model-free learning as well. However, in our preliminary experiments with R-learning, we found that the value of ρ fluctuates much more in R-learning than in H-learning, unless α is initialized to be

very small. However, a small α will have the consequence of slowing down learning. Another possibility is to maintain a confidence interval for ρ and to use the upper bound of the confidence interval to update the R values as in the IE method.

Most reinforcement learning work is based on model-free algorithms such as Q-learning [46]. Model-free algorithms are easier to implement, because they have simpler update procedures. However, it has been observed that they do need more real-time experience to converge because they do not learn explicit action models, which are independent of the control policy and can be learned fairly quickly [3,28]. Once the action models are learned, they can be used to propagate more information in each update of the value function by considering all possible next states of an action rather than the only next state that was actually reached. They can also be used to plan and to learn from simulated experience as in the Dyna architecture [40]. However, one of the stumbling blocks for the model-based algorithms to be more widely used is that representing them explicitly as transition matrices consumes too much space to be practical. Dynamic Bayesian networks have been used by a number of researchers to represent action models in decision theoretic planning [7,12,19,30]. We showed that they can also be useful to compactly represent the action models for reinforcement learning and to shorten the learning time. Our current method uses the structure of the dynamic Bayesian network as prior knowledge and learns only the conditional probability tables. One of the important future research problems is to learn the structure of these networks automatically.

We showed that the piecewise linearity of the h-function can be effectively exploited using Local Linear Regression (LLR). It is a member of a family of regression techniques that go under the name of Locally Weighted Regression (LWR) [2,29]. LWR is a regression technique with a sound statistical basis that also takes into account the locality of the target function. As a result, it can fit functions that are smooth in some places, but complex in other places. There have been many successful applications of LWR in reinforcement learning, including a juggling robot [36]. Our results suggest that local linear regression (LLR) is a promising approach to approximation especially for Average-reward RL. We also showed that it synergistically combines with approximating domain models using Bayesian networks. However, being a local method, it does not scale well with the number of dimensions of the state space, especially when the value function is nonlinear in these dimensions, e.g., the number of AGVs, machines, or the conveyor belts in our domains. It may be necessary to combine it with other feature selection methods, or instead use more aggressive approximation methods like the neural networks or regression trees to scale learning to larger domains [9,11,21]. Another important problem is to extend our work to domains where some features, such as the location of the AGV, can be real-valued.

To apply our methods to the full-scale AGV scheduling, we need to be able to handle multiple AGVs. Since AGVs are still very expensive, minimizing the number of AGVs needed for a given factory floor is an important practical problem. Treating all the AGVs as a single agent does not scale because the set of actions available to the AGV system is the cross-product of the sets of actions of all the AGVs. There have been some positive results in multi-agent reinforcement learning, including Crites's work on scheduling a bank of elevators, and Tan's results in a hunter-prey simulation [11,43].

Our preliminary experiments in a simple domain with 2 AGVs indicate that an optimal policy can be learned as a mapping from global state space to actions of a single AGV. Both AGVs share and update the same value function and follow the same optimal policy. This approach converges faster to a global optimal policy than treating the two AGVs as a single agent.

Another important assumption that we made that needs relaxing is that the state is fully observable. Recently, there has been some work to extend RL to Partially Observable Markov Decision Problems (POMDPs) [22,33]. Unfortunately, even the best algorithms for solving POMDPs currently appear to be impractical for large problems. We believe that building in more domain-specific prior knowledge into the learning algorithms in the form of high-level features or constraints on the value function is essential to further progress in this area.

7. Conclusions

Reinforcement learning has proved successful in a number of domains including some real-world domains. However, reinforcement learning methods that are currently most popular optimize discounted total reward, whereas the most natural criterion in many domains such as the AGV scheduling is to optimize the average reward per time step. We showed that employing discounted RL methods to optimize average reward could lead to suboptimal policies, and is prohibitively sensitive to the discount factor and the exploration strategy. We presented a family of algorithms based on a model-based average-reward RL method called H-learning, and empirically demonstrated their usefulness. We presented an auto-exploratory version of our learning method, which outperforms other previously studied exploration strategies. We showed that our learning method can exploit prior knowledge of its action models in the form of dynamic Bayesian network structure, and can improve both the space and time needed to learn them. We also showed that the value functions of H-learning can be effectively approximated using local linear regression. Several open problems remain, including scaling to domains with large number of dimensions, multi-agent RL, and theoretical analysis. We believe that we laid a foundation to study these problems and to apply average-reward reinforcement learning to a variety of real-world tasks.

Acknowledgments

We gratefully acknowledge the support of NSF under grant number IRI-9520243 and the support of ONR under grant number N00014-95-1-0557. We thank Chris Atkeson, Leemon Baird, Andrew Barto, Dimitri Bertsekas, Tom Dietterich, Leslie Kaelbling, Sridhar Mahadevan, Toshi Minoura, Andrew Moore, Martin Puterman, Jude Shavlik, Satinder Singh, and Rich Sutton for many interesting discussions on this topic. We thank Sandeep Seri for his help with testing our programs. We thank the reviewers of this paper for their excellent suggestions.

References

[1] D.W. Aha, D. Kibler, M.K. Albert, Instance-based learning algorithms, Machine Learning 6 (1991) 37–66.

[2] C.G. Atkeson, A.W. Moore, S. Schaal, Locally weighted learning, Artificial Intelligence Review 11 (1997) 11–73.

[3] A.G. Barto, S.J. Bradtke, S.P. Singh, Learning to act using real-time dynamic programming, Artificial Intelligence 73 (1995) 81–138.

[4] D. Bertsekas, A new value-iteration method for the average cost dynamic programming problem, Technical Report LIDS-P-2307, MIT, Boston, MA, 1995.

[5] D.P. Bertsekas, Distributed dynamic programming, IEEE Trans. Automatic Control 27 (3) (1982).

[6] D.P. Bertsekas, Dynamic Programming and Optimal Control, Athena Scientific, Belmont, MA, 1995.

[7] C. Boutilier, R. Dearden, M. Goldszmidt, Exploiting structure in policy construction, in: Proceedings 14th International Joint Conference on Artificial Intelligence (IJCAI-95), Montreal, Que., 1995.

[8] J. Boyan, A.W. Moore, Generalizing reinforcement learning: safely approximating the value function, in: Proceedings Neural Information Processing Systems, 1994.

[9] L. Brieman, J.H. Friedman, R.A. Olshen, C.J. Stone, Classification and Regression Trees, Wadsworth International Group, Belmont, MA, 1984.

[10] G.C. Canavos, Applied Probability and Statistical Methods, Little, Brown and Company, Boston, MA, 1984.

[11] R.H. Crites, A.G. Barto, Improving elevator performance using reinforcement learning, in: Advances in Neural Information Processing Systems, Vol. 8, MIT Press, Cambridge, MA, 1996.

[12] T. Dean, K. Kanazawa, A model for reasoning about persistence and causation, Computational Intelligence 5 (3) (1989) 142–150.

[13] G.W. Gates, The reduced nearest neighbor rule, IEEE Trans. Inform. Theory (1972) 431–433.

[14] P.E. Hart, The condensed nearest neighbor rule, IEEE Trans. Inform. Theory 14 (1968) 515–516.

[15] R.A. Howard, Dynamic Programming and Markov Processes, MIT Press and Wiley, Cambridge, MA, 1960.

[16] A. Jalali, M. Ferguson, Computationally efficient adaptive control algorithms for markov chains, in: IEEE Proceedings 28th Conference on Decision and Control, Tampa, FL, 1989.

[17] L.P. Kaelbling, Learning in Embedded Systems, MIT Press, Cambridge, MA, 1990.

[18] L.P. Kaelbling, M.L. Littman, A.W. Moore, Reinforcement learning: a survey, J. Artificial Intelligence Research 4 (1996) 237–285.

[19] U. Kjaerulff, A computational scheme for reasoning in dynamic probabilistic networks, in: Proceedings 8th Conference on Uncertainty in Artificial Intelligence (1992) 121–129.

[20] S. Koenig, R.G. Simmons, The effect of representation and knowledge on goal-directed exploration with reinforcement-learning algorithms, Machine Learning 22 (1996) 227–250.

[21] L.J. Lin, Self improving reactive agents based on reinforcement learning, planning, and teaching, Machine Learning 8 (1992) 293–321.

[22] M.L. Littman, A. Cassandra, L.P. Kaelbling, Learning policies for partially observable environments: scaling up, in: Proceedings of International Machine Learning Conference, San Fransisco, CA, 1995, pp. 362–370.

[23] S. Mahadevan, An average reward reinforcement learning algorithm for computing bias-optimal policies, in: Proceedings National Conference on Artificial Intelligence (AAAI-96), Portland, OR, 1996.

[24] S. Mahadevan, Average reward reinforcement learning: foundations, algorithms, and empirical results, Machine Learning 22 (1996) 159–195.

[25] S. Mahadevan, Sensitive discount optimality: Unifying discounted and average reward reinforcement learning, in: Proceedings International Machine Learning Conference, Bari, Italy, 1996.

[26] S. Mahadevan, J. Connell, Automatic programming of behavior-based robots using reinforcement learning, Artificial Intelligence 55 (1992) 311–365.

[27] W.L. Maxwell, J.A. Muckstadt, Design of automatic guided vehicle systems, Institute of Industrial Engineers Trans. 14 (2) (1982) 114–124.

[28] A.W. Moore, A.G. Atkeson, Prioritized sweeping: Reinforcement learning with less data and less time, Machine Learning J. 13 (1993) 103–130.

[29] A.W. Moore, C.G. Atkeson, S. Schaal, Locally weighted learning for control, Artificial Intelligence Review 11 (1997) 75–113.
[30] A.E. Nicholson, J.M. Brady, The data association problem when monitoring robot vehicles using dynamic belief networks, in: ECAI 92: 10th European Conference on Artificial Intelligence Proceedings, Vienna, Austria, Wiley, New York, 1992, pp. 689–693.
[31] D. Ok, A study of model-based average reward reinforcement learning, Ph.D. Thesis, Technical Report, 96-30-2, Department of Computer Science, Oregon State University, Corvallis, OR, 1996.
[32] D. Ok, P. Tadepalli, Auto-exploratory average reward reinforcement learning, in: Proceedings National Conference on Artificial Intelligence (AAAI-96), Portland, OR, 1996.
[33] R. Parr, S. Russell, Approximating optimal policies for partially observable stochastic domains, in: Proceedings National Conference on Artificial Intelligence (AAAI-94), Seattle, WA, 1994, pp. 1088–1093.
[34] M.L. Puterman, Markov Decision Processes: Discrete Dynamic Stochastic Programming, John Wiley, New York, 1994.
[35] S. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, Prentice-Hall, Englewood Cliffs, NJ, 1995.
[36] S. Schaal, C. Atkeson, Robot juggling: an implementation of memory-based learning, in: IEEE Control Systems, Vol. 14, 1994, pp. 57–71.
[37] A. Schwartz, A reinforcement learning method for maximizing undiscounted rewards, in: Proceedings 10th International Conference on Machine Learning, Morgan Kaufmann, San Mateo, CA, 1993.
[38] S.P. Singh, Reinforcement learning algorithms for average-payoff markovian decision processes, in: Proceedings National Conference on Artificial Intelligence (AAAI-94), Seattle, WA, MIT Press, Cambridge, MA, 1994.
[39] R.S. Sutton, Learning to predict by the methods of temporal differences, Machine Learning 3 (1988) 9–44.
[40] R.S. Sutton, Integrating architectures for learning, planning and reacting based on approximating dynamic programming, in: Proceedings Seventh International Conference on Machine Learning, Austin, TX, 1990.
[41] P. Tadepalli, D. Ok, H-learning: A reinforcement learning method for optimizing undiscounted average reward, Technical Report 94-30-1, Department of Computer Science, Oregon State University, 1994.
[42] P. Tadepalli, D. Ok, Scaling up average reward reinforcement learning by approximating the domain models and the value function, in: Proceedings 13th International Conference on Machine Learning, 1996.
[43] M. Tan, Multi-agent reinforcement learning: independent vs. cooperative agents, in: Proceedings 10th International Conference on Machine Learning, Morgan Kaufmann, San Mateo, CA, 1993.
[44] G. Tesauro, Practical issues in temporal difference learning, Machine Learning 8 (3–4) (1992) 257–277.
[45] S. Thrun, The role of exploration in learning control, in: Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches, Van Nostrand Reinhold, New York, 1994.
[46] C.J.C.H. Watkins, P. Dayan, Q-learning, Machine Learning 8 (1992) 279–292.
[47] W. Zhang, T. Dietterich, A reinforcement learning approach to job-shop scheduling, in: Proceedings 14th International Joint Conference on Artificial Intelligence (IJCAI-95), Montreal, Que., 1995.