# Simulation and Animation of Reinforcement Learning Algorithm

Vivek Mehta
Robotics Institute
Carnegie Mellon University
Pittsburgh, PA-15216
Email: vivekmehta@cmu.edu

Rohit Kelkar
Robotics Institute
Carnegie Mellon University
Pittsburgh, PA-15216

*Advisor:*
Andrew Moore
Robotics Institute
Carnegie Mellon University
Pittsburgh, PA-15216

***Abstract*— Simulation and Animation of an algorithm is useful tool to visualize its behavior. It enables analysis and hence improvements to existing algorithms. With this driving force, four classical reinforcement learning (RL) algorithms, namely *value iteration*, *policy iteration*, *Q-learning* and *prioritized sweeping*, were simulated and animated to observe their behavior and to analyze their performance in different experimental setting than one which uses discount factor. Based on the analysis, two enhancements are proposed, one each for *Q-learning* and *prioritized sweeping* algorithms.**

## I. Introduction

Machine learning is study of algorithms which improve performance with experience [1]. Such an algorithm is known as learner and procedure of improving performance with experience is known as training experience. Based on nature of training experience we get three types of machine learning algorithms:

1) *Supervised Learning*: In Supervised Learning, teacher provides a desired response (output) for a given situation (input) and learner is suppose to learn mapping of best response given a situation.
2) *Reinforcement Learning (RL)*: In RL, given a situation (state) and associated action a reward is know to learner. Based on these rewards, learner is supposed to learn the best action given a situation.
3) *Unsupervised Learning*: In unsupervised learning, learner tries to learn the pattern in data representing different situations (input).

This report will focus on RL and related algorithms. Rest of the report is organized as follows: In Section II, RL problem is formalized. The experimental setup (model), used for simulation and animation of RL algorithms, is described in Section III. Section IV and V describe algorithms adapted to the model given in III, when model is known and unknown respectively. In Section V, two enhancements to existing algorithms are presented. Section VI talks about future work and conclusion.

The two authors collaborated on the experimental portion of this work but this report was written solely by first author.

## II. Formalizing RL Problem

### A. Example

Consider an autonomous room cleaning robot, which is required to clean the entire room, without hitting any obstacle and without getting its battery discharged completely. In case, the robot is low on battery it should return to battery charging station before battery gets discharged completely.

One way to solve this problem is to manually code rules for exploring and cleaning room, avoiding obstacle and checking battery level against distance to recharging station and deciding, when robot should return for recharging. But this approach is complex to implement and not easy to scale.

Alternatively, we can formulate problem in different manner to solve it. The Robot get positive reinforcement for cleaning house, negative reinforcement for hitting obstacle and negative reinforcement if battery gets discharged completely before robot reaches recharging station. Then, robot can learn the best strategy, to clean room by moving around in the room and trying to maximize the reinforcement it receives. This strategy is easy to implement and scalable also. This way of solving problem is know as RL.

### B. Formal RL Model

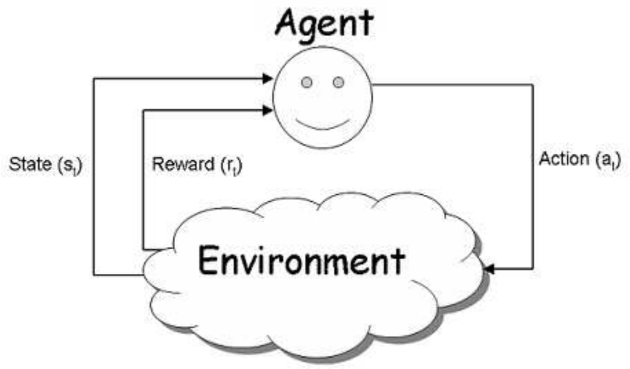

Fig. 1. RL model: Agent-Environment Interface

As explained in above example, RL is learning from interaction with environment. The learner and decision-maker is called the *agent*. The thing it interacts with, comprising everything outside the agent, is called the *environment*. These interact continually, the agent selecting actions and the environment responding to those actions and presenting new situations (known as states) and reward associated with them to the agent, as shown in Figure 1.

There are two flavors of RL problem based on the fact, that the agent is aware of the environmental model or not. Algorithms in both these scenarios are discussed in this report.

### C. *Policy*

Policy ($\pi(s)$) is mapping of states to actions. An optimal policy ($\pi^*(s)$), is one which have mapping of states to best action. Finding optimal policy is heart of all RL problems.

## III. Rules of the Game

For this experiment, a maze-world is used with every cell representing a state. There are one or more goal cells. There can be a wall present on the boundary between any two adjacent cells. Agent occupies cell completely at any given instant. Agent can take four actions: UP, DOWN, LEFT, RIGHT, which tries to take it to the adjacent state in corresponding direction. It is not possible for agent to cross the walls and it gets penalty if it hits a wall. Also every step taken in maze costs and agent gets penalty equivalent to path cost. Agent get zero reinforcement on reaching any goal state.

Noise in environment is modelled using concept of PJOG. Environment is associated with PJOG which can take value between 0 and 1. PJOG affects agents action as follows: If agent wants to take action UP, then agent take that action with probability (1-PJOG) and it takes one of remaining action with probability PJOG/3.

Given starting state, agent is supposed to reach goal with as low penalty as possible i.e. agent is supposed to find optimal policy for given maze.

This model is different from models which are generally used in RL problems [2] [3], as in this model, agent does not get any reward. It gets either zero or positive penalty as reinforcement.

## IV. Learning an Optimal Policy: Model Known

If model is known, Markov Decision Processes(MDPs) can be used to represent the problem. Then, problem of finding optimal policy can be solved using matrix inversion method [4], but its time complexity is $O(N^3)$. Better algorithms exist [2] [3] which use concepts of Dynamic Programming (DP).

### A. *Markov Decision Processes*

MDP is used to model reinforcement learning problems. It consists of:

- a set of states, $S$
- a set of actions, $A$
- a reward function, $R : S \times A \rightarrow \Re$, and
- a transition probability function, $P(s'|s,a)$, where, $s', s \in S$ and $a \in A$

It can be observed that, the model described in Section III can be directly mapped to an MDP.

### B. *Dynamic Programming*

Dynamic programming can be used to obtain efficient algorithm for solving RL problem. The basic idea is to divide big dynamic problem into small static problems, which repeat multiple times. Solution to such small problems is independent of outer problem. Complete solution for the problem can be obtained by combining solution to smaller problems.

This is the concept on which the *value iteration* and *policy iteration* algorithms are based. We will, now define one more concept of value function, which is used for applying dynamic programming to RL problem.

### C. *Value Function*

Value function is function of states (or state-action pair) that estimates how good it is for the agent to be in a given state (or how good it is to perform a given action in a given state). The goodness of state depends of future rewards that can be expected.

### D. *Value Iteration*

In *value iteration* [2] [3], first optimal value function is computed and then based on that optimal policy is derived. To find optimal value function we define $V_k(S_i)$ as "Maximum possible future sum of rewards that can be obtained, starting at state $S_i$ in k time steps". Then $V_1(S_i)$ will be immediate reward and $V_2(S_i)$ will be function of immediate reward and $V_1(S_i)$. On extending this concept further we can express, $V_{t+1}(S_i)$ for the given experimental setup as follows:

$$V_{t+1}(s) = \min_{a \in A} \left( PCost + \sum_{s'=S} (P(s'|s,a) \cdot x_{ss'}) \right)$$

where,

PCost = PathCost

$P(s'|s,a)$ = Prob. of transition from $s$ to $s'$ after action $a$.

$x_{ss'} = V_t(s')$, if transition from $s$ to $s'$ is safe

$= Penalty + V_t(s)$, if transition from $s$ to $s'$ is not safe

Above step is repeated until value function gets converged and there is not any significant change in values. Since the goal is to minimize reinforcement, optimal policy in any given state is that action, which takes to the state with least value of value function.Thus, policy can be determined as follows:

$$\pi^*(s) = \arg\min_{a \in A} \left( \sum_{s' \in S} (P(s'|s,a) \cdot x_{ss'}) \right)$$

Complete *value iteration* algorithm adapted to given experimental setup is given in Algorithm 1:

### E. *Policy Iteration*

The *policy iteration* algorithm [2], manipulates policy directly instead of finding it via the optimal value function. The basic steps of algorithm are as follows:

1) Choose policy arbitrarily
2) Evaluate policy
3) Update policy

**Algorithm 1** Value Iteration

```
1: initialize $V(s)$ arbitrarily
2: repeat
3:   for all $s \in S$ do
4:     $V_{t+1}(s) = \min_{a \in A} \left( PCost + \sum_{s' \in S} P(s'|s,a) \cdot x_{ss'} \right)$
5:   end for
6: until $V(s)$ has converged
7: for all $s \in S$ do
8:   $\pi^*(s) = \arg\min_{a \in A} \left( \sum_{s' \in S} (P(s'|s,a) \cdot x_{ss'}) \right)$
9: end for
```

4) If not optimal policy, goto 2

Modified *value iteration* step can be used for evaluating a policy and finding value function $V(s)$ for given policy $\pi(s)$ as follows:

```
repeat
  for all $s \in S$ do
    $$V_{t+1}(s) = PCost + \sum_{s' \in S} (P(s'|s,\pi(s)) \cdot x_{ss'})$$
  end for
until $V(s)$ has converged
```

where,
$P(s'|s,a)$ = Prob. of transition from $s$ to $s'$ after action $a$.
$x_{ss'} = V_t(s')$, if transition from $s$ to $s'$ is safe
$= Penalty + V_t(s)$, if transition from $s$ to $s'$ is not safe

Value function computed in above step to evaluate a policy can be used for updating policy. New policy can be determined in similar way as it was done in *value iteration* algorithm. Thus complete policy iteration algorithm, adapted to given experimental setup, is given in algorithm 2.

**Algorithm 2** Policy Iteration

```
1: initialise $\pi(s)$ arbitrarily
2: repeat
3:   repeat
4:     for all $s \in S$ do
5:       $V_{t+1}(s) = PCost + \sum_{s' \in S} (P(s'|s,\pi(s)) \cdot x_{ss'})$
6:     end for
7:   until $V(s)$ has converged
8:   for all $s \in S$ do
9:     $\pi_{t+1}(s) = \arg\min_{a \in A} (\sum_{s'=S}(P(s'|s,a) \cdot x_{ss'}))$
10:  end for
11: until policy good enough
```

## F. Analysis

*Policy iteration* takes less number of iteration to converge than *value iteration* for a given maze, but time taken by *policy iteration* need not be less than that by *value iteration*. Performance of these algorithms depends on ratio of number of actions to number of states. Higher the ratio better will be performance of *policy iteration*.

## G. Modified Policy Iteration

*Policy iteration* converges with very few number of iterations, but every iteration takes much longer time than an iteration of *value iteration*. The main reason for this is the evaluate policy step of *policy iteration*. Solving to find exact value of $V(S)$ for given policy is very expensive, as the change in $V_{t+1}(s)$ becomes very small as $t$ increases. Instead of finding exact value of $V(s)$ for given policy $\pi$, a few steps of value-iteration can be performed such that change in value function in not significant (less than a small threshold). Thus, the evaluate policy step of *policy iteration* gets modified as follows to give rise to *modified policy iteration*:

```
repeat
  for all $s \in S$ do
    $$V_{t+1}(s) = PCost + \sum_{s' \in S} (P(s'|s,\pi(s)) \cdot x_{ss'})$$
  end for
until change in $V(s)$ is not significant
```

## V. Learning an Optimal Policy: Model Unknown

Algorithms discussed in previous section were based on assumption that the model is known and can be represented as a MDP. Model is known essentially means that state transition probability $P(s'|s,a)$ and reinforcement function $R(s,a)$ is known. There are lot of practical problems in which model is not known. In such situations, the agent must interact with its environment to obtain information which can be processed to determine the optimal policy. Such algorithms will be discussed in this section.

When model is not known, there are two types of algorithms which are possible:

1) *Model-free:* Learn policy without learning model.
2) *Model-based:* Learn a model, and use it to derive a policy.

Both types of algorithms are being used in practice. In this report, one algorithm of each type, *Q-learning* (Model-free) and *prioritized sweeping* (model-based), is discussed.

## A. Q-learning

Watkins' *Q-learning* [2] is a model-free method which is very easy to implement. Q-learning works by estimating the value of state-action pair, $Q(s,a)$. The value $Q(s,a)$, known as Q-value, is defined to be the expected sum of future reinforcement(penalty) obtained by taking action $a$ from state $s$ and following an optimal policy thereafter. Once these values have been learned, the optimal action from any state is the one with the lowest Q-value. So, if $Q^*(s,a)$ is optimal Q-values, optimal policy can be obtained as follows:

$$\pi^*(s) = \arg\min_{a \in A} (Q^*(s,a))$$

The estimation of Q-values can be done on the basis of experience using following learning rule:

$$Q(s,a) = Q(s,a) + \alpha \left( X + \min_{a' \in A} (Q(s',a') - Q(s,a)) \right)$$

where,

$s'$ is new state after taking action $a$ on state $s$ and $X$ is reinforcement observed.

$X = PathCost$ , if transition is safe

$X = R$ , if transition is unsafe

$\alpha$ is known as learning rate.

As shown in learning rule, for every observation we update Q-value based on observation. Learning rate determines how much current Q-value is changed on the basis of new observation. Learning rate can take value between 0 and 1. In noisy environment, high learning rate would not allow the system to stabilize and low learning rate makes learning very slow. Thus decaying learning rate is used in such case.

For Q-values to converge to optimal values, it is required that every state and action pair is explored sufficient number of times, ideally infinite number of times. Thus an exploration policy must be used for choosing an action given a state.

Some of exploration policies used for this experiment are:

- $\epsilon$-greedy: Select best action with probability $1-\epsilon$ and choose random action with probability $\epsilon$.
- Changing start state: Once goal is reached, reset to random state for next iteration.
- Time-worn virtual reward: Artificially give high virtual rewards for state and action which have not been explored for a long time.

The complete Q-learning algorithm using $\epsilon$-greedy exploration policy and adapted to given experimental setups is given in algorithm 3.

**Algorithm 3** Q-learning

```
1: initialise $Q(s,a)$ arbitrarily
2: loop
3:   initialize $s$
4:   repeat
5:     select action $a$ from $s$ based on $Q(s,a)$ using $\epsilon$-greedy policy
6:     take action $a$, Observe reward $r$, next state $s'$
7:     $Q(s,a) = (1-\alpha)Q(s,a) + \alpha[X + min_{a'}Q(s',a')]$
8:   until $s$ is goal
9: end loop
```

*Note:* Step 4 to 8 is know as episode.

In following subsections, effect of learning rate on Q-learning is analysed and adaptive learning rate is proposed for non-stationary environment.

*1) Decaying learning rate:* According to this scheme the agent interacts with the environment with a very high learning rate initially so that it can quickly settle down to a near optimal policy. As the agents interactions with the environment increase the learning rate decays and the agent now makes only minor modifications to its near optimal policy to obtain the optimal policy. This scheme is guaranteed to asymptotically converge to the optimal policy. The learning rate under the decaying learning rate scheme is shown in Figure 2. The results of using the decaying learning rate are shown in Figure

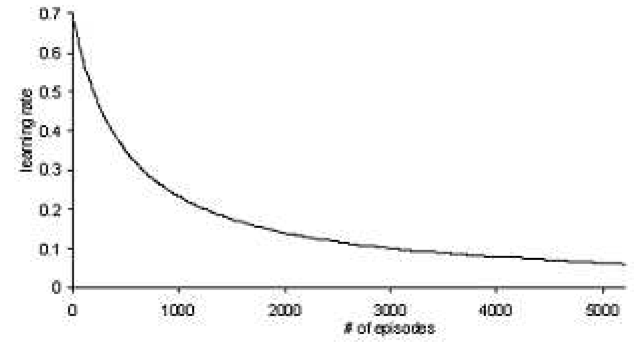

Fig. 2. Decaying learning rate

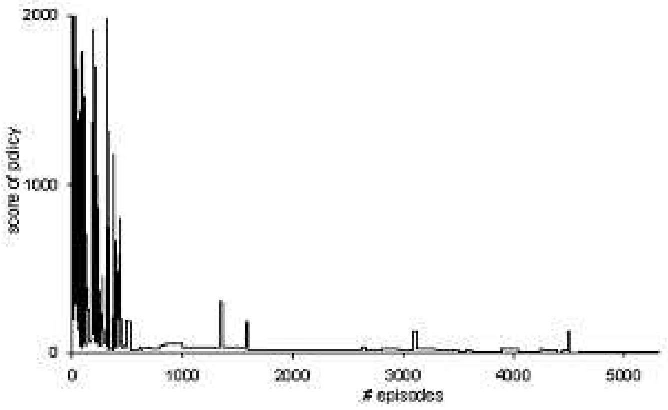

Fig. 3. Score of policy for decaying learning rate

3. It can be seen that the score [1]of the policy decays faster due to high learning rate in the initial stages and then as the number of episodes increase the learning rate decays further and the score of the policy drops to zero.

*2) Decaying learning rate applied to non stationary environment:* When the environment is non stationary the use of decaying learning rate causes degradation in the performance of the algorithm. As number of episodes increase the learning rate becomes very small and the changes in the environment does not make significant changes in the Q-values that the agent maintains. Because of this the agent keeps executing a sub-optimal policy. This phenomenon can be seen in the Figure 4. Here the environment is changed after every 2000 episodes. It can be observed at one such instance after 6000 episodes the environment changes and the score of the policy that the agent is executing takes a long time to settle to zero. This means that the agent takes longer time to discover the optimal policy. It has also been observed that as the agents interactions with

[1]The score of the policy is computed as follows:

$$Score(\pi) = \sigma||V^{\pi}(s) - V^{*}(s)||$$

where,

$\pi$ = Learned policy whose value is to be evaluated

$\pi^{*}$ = True optimal policy

$V^{\pi}(s)$ = True value for using policy $\pi$ starting at state s

$V^{*}(s) = V^{\pi}(s)$

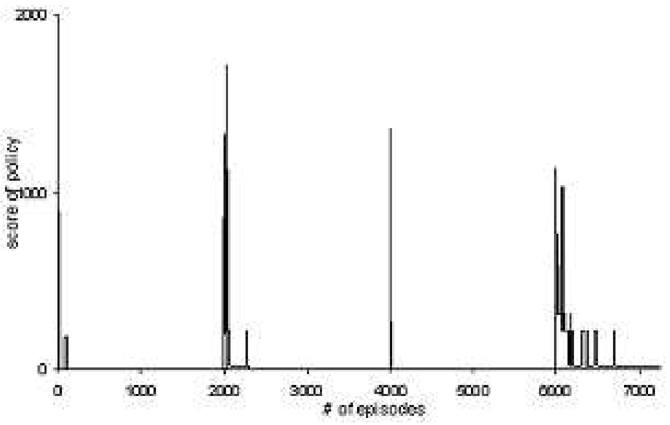

Fig. 4. The score of policy for changing environment with decaying learning rate

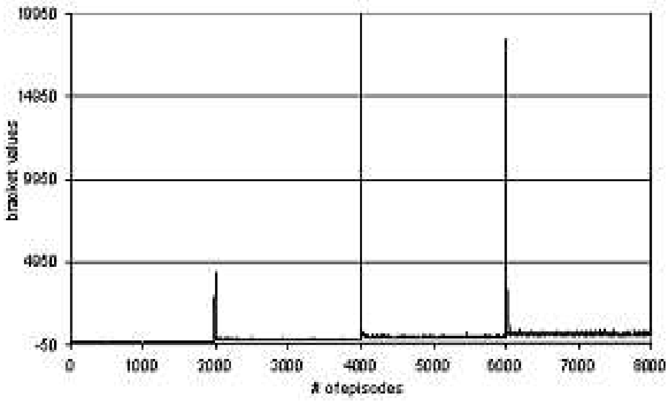

Fig. 5. The bracket values as the number of episodes increase for decaying learning rate scheme

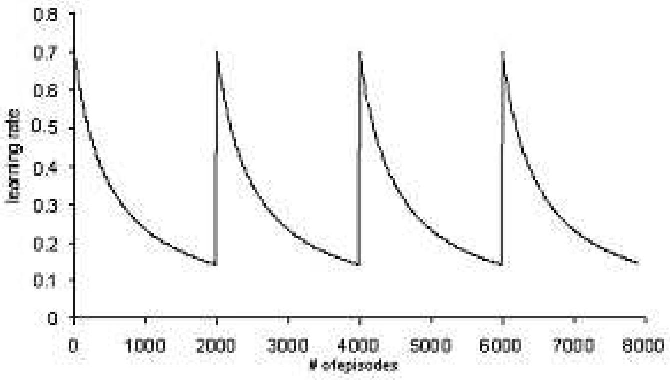

Fig. 6. Adaptive learning rate

the environment increase the learning rate further decreases and each time the environment changes the time taken by the agent to discover the optimal policy increases.

It is clear from the above discussion that the problem of increased time taken by the agent to find the optimal policy can be attributed to extremely low learning rate as a result of using the decaying learning rate scheme. At the same time the decaying learning rate scheme cannot be discarded because it undoubtedly performs better then the fixed learning rate scheme.

*3) Adaptive learning rate in non stationary environments:* To remedy above problem with the decaying learning rate scheme as applied to non stationary environments the concept of adaptive learning rate is introduced. This involves resetting the learning rate whenever the environment changes. But this requires the agent to infer intelligently that the environment has changed based on the stimuli that it receives from the environment. One such indication that the environment has changed is the sudden rise in the values that the agent uses to update its Q-values. Consider the Q-learning learning rule:

$$Q(s,a) = Q(s,a) + \alpha(X + min_{a'}Q(s',a') - Q(s,a))$$

$$Q(s,a) = Q(s,a) + \alpha(Bracket - Value)$$

where,
Bracket-Value = $(X + min_{a'}Q(s',a') - Q(s,a))$

Bracket-Value was observed for 8000 episodes with environment changing at every 2000 episodes. From Figure 5, it is clear that whenever the environment changes there is a sudden increase in the Bracket-Value.

Thus Bracket-Values are good indicators of the changes in the environment. If the learning rate in the decaying learning rate scheme is reset each time the environment changes then such a scheme would perform with the same advantages as the decaying learning rate scheme in non stationary environments as well. So the problem now is to detect the changes in the Bracket-Values. This can be done by maintaining a window of past n Bracket Values and detecting a change in the means of two successive windows. If the environment changes the change in means between two successive windows is significant. To detect the change in means the t-test is used as follows:

$$t = \frac{X_i - X_{i-1}}{\sqrt{\frac{var_i}{n} - \frac{var_{i-1}}{n}}}$$

$$t > 3 \times \sqrt{n} \Rightarrow Environment\ changed$$

where,
$n$ : Window size
$X_i$ : Mean of window after $i^{th}$ episode
$var_i$: Variance of window after $i^{th}$ episode

The results of using the adaptive learning rate with Q-learning in non stationary environments are shown in Figure 6 and 7. It can be clearly seen that by resetting the learning rate when the environment changes the agent is able to find the optimal policy in a shorter period of time as compared to the decaying learning rate.

## *B. Prioritized Sweeping*

*Prioritized sweeping* [5] [3] is a smart model-based algorithm, which explore interesting part of state-space by using information obtained from experience. It uses and updates

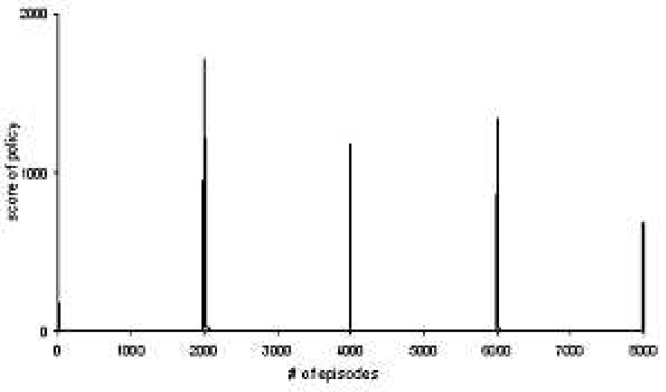

Fig. 7. Adaptive learning rate

values associated with state instead of state-action pairs (as in Q-learning).

The basic idea of this algorithm is that when agent encounters a transition which is surprising (i.e. transition changes value function of current state by significant amount), this information is propogated to relevant predecessor states (also know as backups). When the transition is boring (i.e. new value of value function is same as expected value) computation continues in more deserving part. In order to build model and make appropriated choices, following information are stored:

- Statistics for the transition from $s$ to $s'$ on action $a$. This information is used to estimate transition probability, $P(s'|s,a)$.
- Statistics for reinforcemnt recieved for taking action $a$ in state $s$.
- Every states predecessor: the state that have non-zero transition probability to it under some action.

Using the model build from experience, algorithm estimates state's value $\hat{V}(s)$ using value iteration like update rule.

$$\hat{V}(s) = \min_{a \in A} \left( \hat{r}_i^a + \sum_{s' \in S} \hat{q}_{ss'}^a \hat{V}(s') \right)$$

where,

$\hat{V}(s)$: Estimate of optimal reward starting from state $s$

$\hat{r}_i^a$: Estimate of immediate reinforcement

$\hat{q}_{ss'}^a$: Estimate of transition probability from state $s$ to $s'$ with action $a$

*prioritized sweeping* algorithm, based on above equation, is given in algorithm 4.

*1) Optimal Backups: Prioritized sweeping* uses careful bookkeeping to concentrate all computational effort on most interesting parts of the system. But there is one question unanswered "How many backups will give the optimal performance?" Intuitively, a very small number of backups or very high number of backups will not give optimal performance. In order to find the precise answer to this question, experiments were performed on Prioritized Sweeping algorithm with varying number of backups and fixing other parameters. This experiment was carried out for both the cases of noisy as well as noiseless environment.

**Algorithm 4** Prioritized Sweeping

```
1: loop
2:   repeat
3:     Take action $a$ from current state, observe new state and reward.
4:     Update model with observed information.
5:     Promote recent state to top of priority queue
6:     while Number of backups processed is less than allowed and priority queue not empty do
7:       Remove the top state from priority queue. Call it s.
8:       $V_{old} = \hat{V}(s)$
9:       $\hat{V}(s) = \min_{a \in A} \left( \hat{r}_i^a + \sum_{s' \in S} \hat{q}_{ss'}^a \hat{V}(s') \right)$
10:      $\Delta = |V_{old} - \hat{V}(S)|$
11:      for each $(s', a') \in preds(s)i$ do
12:        $P = \hat{q}_{s's}^{a'} \Delta$
13:        if ($P > \epsilon$ (a tiny threshold)) and ($S'$ not on queue or $P$ exceeds the current priority of $s'$ ) then
14:          Promote $s'$ to new priority $P$
15:        end if
16:      end for
17:    end while
18:  until $s$ is goal
19: end loop
```

**Noiseless Environment:** Following parameters were used first for experiment with noiseless environment:
*Maze*: 45x45 (state-space size: 2045),
*PJOG*: 0 (No noise),
*Exploration policy*: $\epsilon$-greedy with $\epsilon$ = 0.1
*Number of backups used*: 1, 5, 25, 125, 625, 2045

Corresponding error curve, where error is nothing but score of policy, with respect to time is shown in fig. 8. It can be observed that for small value of backup (1, 5 and 25) performance was not as good as compared to that for higher values of backups (125, 625 and 2045).

Next experiments was performed with equi-spaced backups 0, 405, 810, 1215, 1620. Corresponding error curve with respect to time is shown in fig. 9. It can be be observed that except for 0 backups all other backups give similar performance. Thus, it can be concluded that for noiseless environment initially performance increases as the number of backups increase but after certain number of backups, increase in number of backups does not improve performance of algorithm.

The reason for this is initially increase in number of backups, helps algorithm explore interesting areas, but after a certain number more backups give same effect. This is because, as there is no limit on size of priority queue, all states which are above tiny threshold are pushed Now if number of backups is high, all the states in queue are processed and queue becomes empty at interesting step. If next step

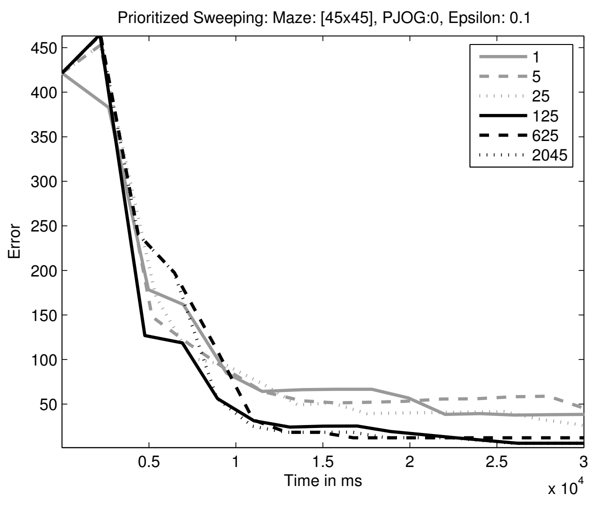

Fig. 8. Error curve I for noiseless environment

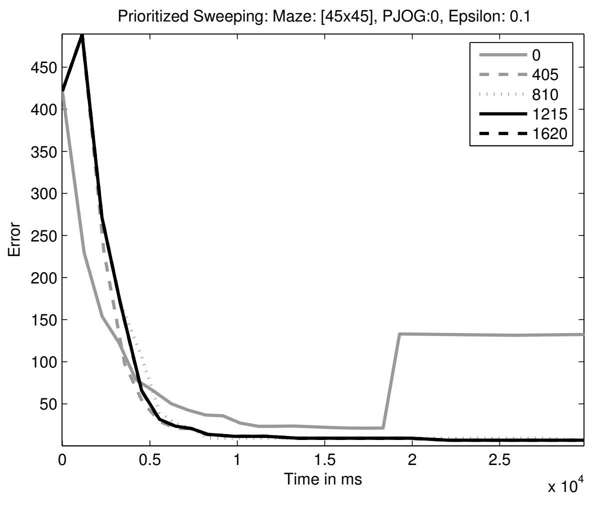

Fig. 9. Error curve II for noiseless environment

is not interesting there is no processing to do. In case of fewer backups, only small number of states are processed at interesting step but remaining states are processed when some uninteresting state is encountered. So at the end of episode there is not much difference between time taken by Prioritized Sweeping with high enough backups or very high backups for noiseless environment.

**Noisy Environment:** Following parameters were used for experiment with noisy environment:

*Mazes*: 10x10 (state-space: 100), 45x45 (state-space: 2045)
*PJOG*: 0.3 (Noise)
*Exploration policy*: Changing start state, $\epsilon$-greedy ($\epsilon = 0.1$)

Figure 10, 11, 12 and 13, shows error curves for different mazes with various number of backups.

It can be observed that performance of algorithm is not very good for very small number of backups (0, 2 for 10x10 maze and 1, 5 for 45x45 maze). At the same time, performance

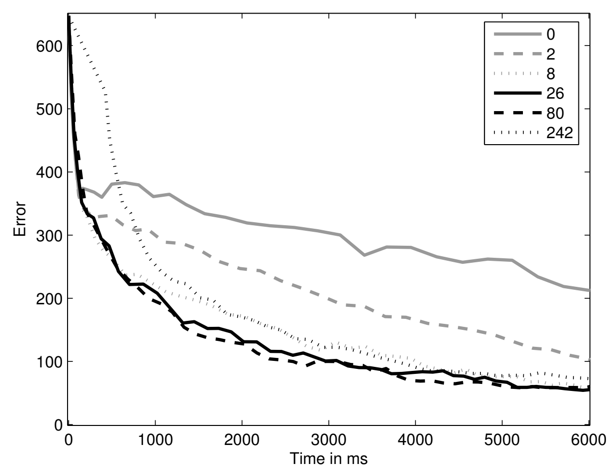

Fig. 10. Error curve I for 10x10 Maze

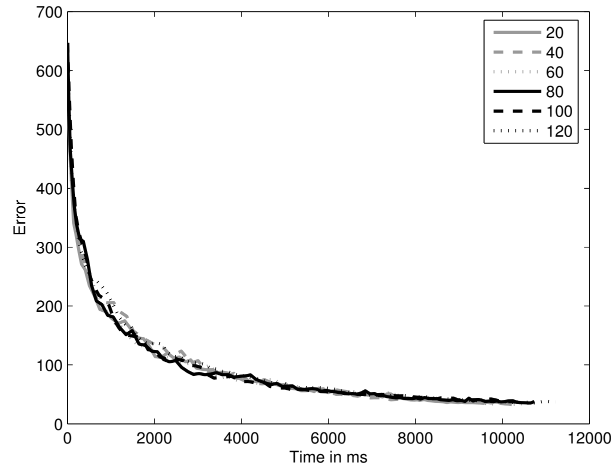

Fig. 11. Error curve II for 10x10 Maze

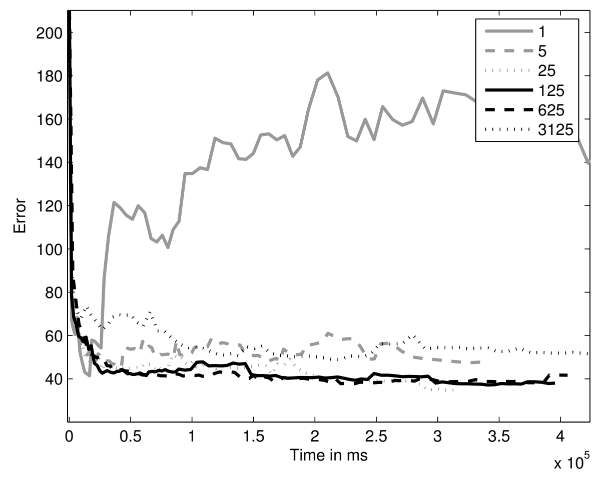

Fig. 12. Error curve I for 45x45 Maze

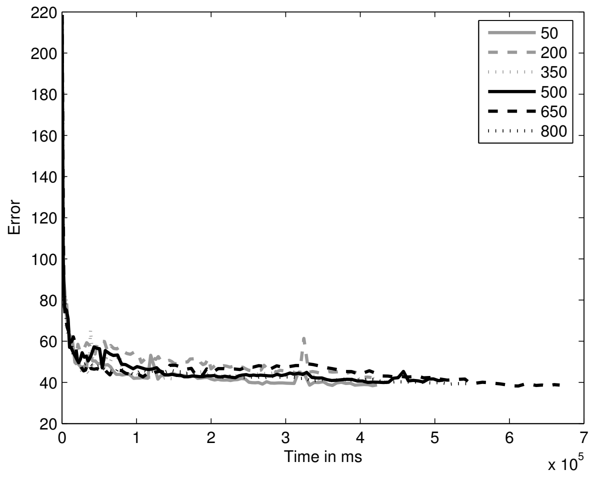

Fig. 13. Error curve II for 45x45 Maze

is not very good for very high number of backups (242 for 10x10 and 3125 for 45x45 maze). But there exists a band of intermediate values for the number of backups allowed which give optimal performance.

## VI. Conclusion

Simulation and animation of an algorithm indeed helps analysing an algorithm and doing enhancement to it. The simulation was used to analyze effect of learning rate on *Q-learning* algorithm and investigate optimal number of backups for *prioritized sweeping*. Adaptive learning rate scheme was proposed for *Q-learning* in non stationary environments and preliminary results are reported for optimal number of backups for *prioritized sweeping*. Also, animation was used to teach RL algorithms.

This work can be extended to enhance animation for more deeper understating of algorithms and more algorithms can be added to system. Work on *prioritized sweeping* can be extended by using actual process time computation and by using bigger and complicated mazes.

## Acknowledgment

The authors would like to thank Prof. Andrew Moore and Prof. Mel Siegel for their advice and constant motivation. We would also like to thank all MSIT-RT students and for their inputs and suggestions. Thanks to Jean-Francoius Lalonde for giving useful suggestions and help on report writing. In the end, thanks to Rachel Burcin and Kristen Schrauder for helping us with administrative work.

## References

[1] T. M. Mitchell, *Machine Learning*. McGraw Hill, 1997.

[2] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. London, England: The MIT Press, 1998.

[3] L. P. Kaelbling, M. Littman, and A. Moore, "Reinforcement learning: A survey," *Journal of Artificial Intelligence Research*, vol. 4, pp. 237–285, 1996.

[4] A. Moore, "Reinforcement learning: Tutorial slides." [Online]. Available: http://www-2.cs.cmu.edu/ awm/tutorials/rl06.pdf

[5] A. Moore and C. Atkeson, "Prioritized sweeping: Reinforcement learning with less data and less real time," *Machine Learning*, vol. 13, pp. 103–130, 1993.