Intelligent Racecars:

Comparing the effectiveness of Q Learning and Value Iteration in a race track setting

Derek Reimanis

December 5, 2012

Abstract

This paper analyzes the use of two algorithms, Q Learning and Value Iteration, in a race track setting. A race car is initialized with one of these two algorithms and told to reach the finish line. Each movement has, in addition to a negative cost, only a chance of being executed successfully. The racer is tested across a multitude of tracks with different shapes, different learning constants, and various punishments for not staying on track. The time in which the racer took to finish is recorded, as well as number of movements, consistency of path, and how close to optimality the path chosen was.

1 Introduction

Artificial Intelligence, that is, an 'intelligent' problem solver, generally revolves around the idea of a model in which an agent and an environment exist. The agent can perceive the environment using its percepts. An agent can also take actions that alter the environment using its actuators. In combination, an agent perceives the environment and returns an action that alters the environment. In what is referred to as a reflex-based model, an agent will apply the current state of the environment to a set of internal rules, thus modeling a virtual environment. This is done without taking any actions upon the actual environment, allowing the agent to effectively 'think' about future environment states. An agent can now plan ahead, using iterations of a virtual environment state to eventually reach a goal state. This is referred to as a goal-based agent; that is, an agent that applies the rules of the environment to a virtual copy of the environment to eventually reach a desired state (Stuart Russell, 2010).

In the process of having an agent learn the best path a goal state, there are a few different methods to use. The first method typically presented is that of 'supervised learning'. Supervised learning means that the correct goal state is known, and the agent merely checks to see if it has made progress towards that goal state after an action. The defining point in supervised learning is that the goal state is known a priori, and the agent tries to optimize the correct solution. A second method presented in agent learning is referred to as 'un-supervised learning'. As the name would suggest, un-supervised learning is the process of moving through a problem, yet the agent knows of no solution to this problem. The learning agent therefore has no method of determining if it is in fact progressing, or even regressing, away from the solution. A final learning method an agent can utilize is that of reinforcement learning. Reinforcement learning acts as a middle ground between supervised and unsupervised learning; that is, reinforcement learning ranks the agent's actions based on some utility, such that the agent can progress through a problem. Now, there may or may not be an answer to this problem, yet the agent does not care. The reinforcement learning agent is only concerned with maximizing a utility, which will hopefully end up in a solution state.

Reinforcement learning is a practical field in Artificial Intelligence. Reinforcement learning is useful when a problem arises that is too difficult to solve without the use of computers, yet a heuristic of a solution can be formulated. Reinforcement learners can be either model-based or model-free learners. Model-based learners are implementations of an agent in which the agent knows how the world works (Stuart Russell, 2010). An agent fully knows the result of applying some action to the environment. A model-free learner, on the other hand, is a learner that has no clue as what the actions it can do are. It may have knowledge of the environment, yet lacks the knowledge of what happens when an action is applied. Typically, a model-free learner must first explore the environment using its actions before a moderately good solution can be found. The difference between the two is that we cannot always assume a model-based learner. In such environments as the relatively un-mapped plains of Mars, a robot may have to learn the environment before a mission can be reliably started.

The algorithms covered in this paper are the Value Iteration and Q Learning algorithms. The Value Iteration algorithm is a model-based learning algorithm, in which the result of applying some transition is known. The Q Learning algorithm, on the other hand, is a model-free learning algorithm. The racecar learner does not know where its actions will put it, so it must experiment with the track in order to learn.

In section 2 of this paper, we describe the setting in which the two algorithms will be applied. The algorithms themselves are discussed, including benefits and detriments to each. In section 3, the basis for experiments performed with Value Iteration are explained. The results of these experiments are covered in detail in section 4, followed by a discussion of the meaning of said experiments in section 5. Finally, in section 6, we look at the future work of implementing the Q Learning algorithm, as well as discuss possible similarities between Value Iteration and Q Learning.

2 Reinforcement Learning on a Racetrack

Now that we have specified exactly what it is that reinforcement learning constitutes, we can look at our model. In this racetrack problem, we start with a racer. The racer has a position, a velocity, and acceleration in a classic Cartesian coordinate system. The laws defining how these values are expressed are governed by the standard law of kinematics, and are expressed below:

$$\begin{aligned}
&x_t \equiv x\ position \\
&y_t \equiv y\ position \\
&\dot{x}_t = x_t - x_{t-1} \equiv x\ speed \\
&\dot{y}_t = y_t - y_{t-1} \equiv y\ speed \\
&a_{xt} = \ddot{x}_t = \dot{x}_t - \dot{x}_{t-1} \equiv x\ acceleration \\
&a_{yt} = \ddot{y}_t = \dot{y}_t - \dot{y}_{t-1} \equiv y\ acceleration
\end{aligned}$$

At any point in time, the racer only has active control over acceleration. Acceleration modifications propagate through velocity and then position, thus moving the car at a rate of acceleration + velocity$_{\text{old}}$ + position$_{\text{old}}$. Acceleration and velocity are limited to create a more realistic model of an actual racer. The control over acceleration values and velocity values is expressed as:

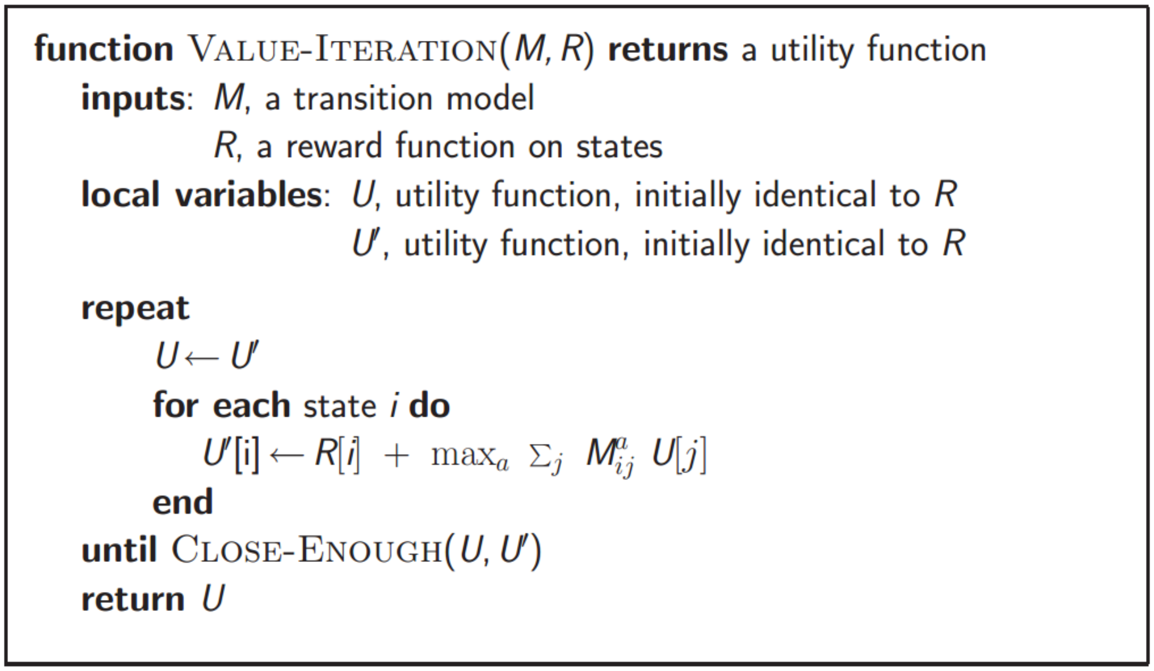
$$\begin{gathered}
a_{\{x,y\}} \in \{-1, 0, 1\} \\
(\dot{x}_t\ , \dot{y}_t) \in [\pm 5, \pm 5]
\end{gathered}$$

To encourage realism, restrictions are placed on acceleration. Put simply, every acceleration applied has a .8 probability of succeeding. This means that 20% of the actions applied will result in an acceleration of (0, 0), meaning the racecar coasts with its current velocity.

To further restrict the racecar's freedom, we penalties to hitting a boundary. The first penalty applied entails that any wall that is hit zeroes the racecar's velocity. The racecar's position is set to closest position to the boundary wall it hit. A second penalty presented states that when a boundary wall is hit, the racecar's velocity is reset to zero with the addition of starting over at the start point. In the case of the second penalty, the racecar must precisely control acceleration vectors to ensure it does not crash, while still maintaining the successful movement probability.

2.1 Value Iteration

The first algorithm the agent will use to navigate the track is the Value Iteration algorithm. The Value Iteration algorithm is a utility based algorithm – that is, it loops through all possible states and actions, generating utility values for each action applied at each state. These utility values are based on rewards for finding the finish line. Value Iteration loops multiple times, thus allowing successive updates of utility values based on neighboring states. Eventually, the algorithm will reach an equilibrium, where no more changes to utilities can be propagated through. When this stage is reached, a vector of utilities is returned. This will effectively tell the racecar which action to apply at any state. Value Iteration is shown below, in Figure 1:



```
function VALUE-ITERATION($M, R$) returns a utility function
    inputs: $M$, a transition model
            $R$, a reward function on states
    local variables: $U$, utility function, initially identical to $R$
                     $U'$, utility function, initially identical to $R$

    repeat
        $U \leftarrow U'$
        for each state $i$ do
            $U'[i] \leftarrow R[i] + \max_a \Sigma_j M^a_{ij} U[j]$
        end
    until CLOSE-ENOUGH($U, U'$)
    return $U$
```

Figure 1 -- The Value Iteration Algorithm (Australia)

Value Iteration starts by taking in input as a transitional model. Very often, this is a Markov Decision Process. A Markov Decision Process, or MDP, is a graph consisting of all possible states as nodes, and actions as edges. Taking an action in a state will lead the agent to a new state. Value Iteration also inputs a reward function for states, specifying the utility of a state. In the racetrack problem, any finishing state will have the highest reward.

Stepping through the algorithm, first note the outer-most loop. This loop specifies when to stop iterating through utility updates. As the change in the utility function through iterations approaches zero, the close-enough function returns true. This is based on the idea that given a vast surplus of time and computing power, Value Iteration will eventually find a (very) close to optimal path. But, realistically, a heuristic function will still result in a solution.

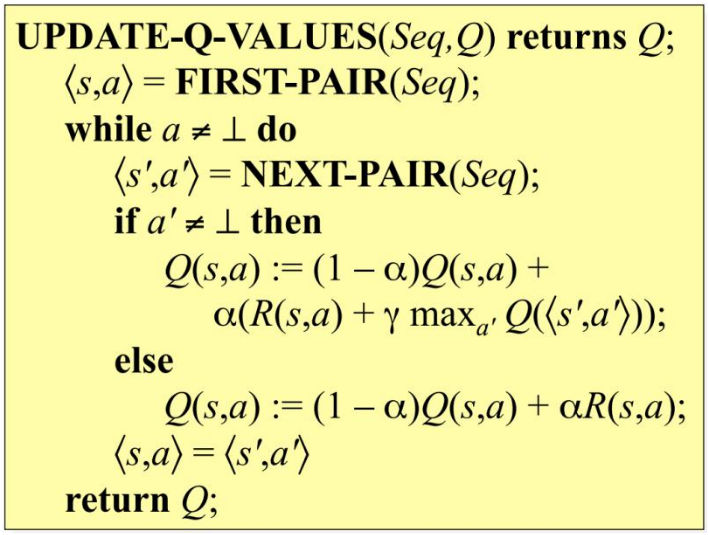
Inside the outer loop is a loop through all the states in the MDP. This loop is basically checking all states, updating their utility values with an augmentation of the utility of the next state. Typically, as in true in our case, there is a cost per move of value -1. Every successive state that is not the finish line will have a decreasing utility, because the cost augments a perfect path. This means that the finish state will have the highest utility, and all other states will have lower utilities based on number of actions away from the finish state.

Inside the state loop is an assignment that takes the max action over a summation of next states and their utilities. Value Iteration's transitions are based on the idea of expected value. This means that a greedy action can be chosen based on the transition states. In any state, all the actions available for that state are applied. The action that returns the maximum expected value is the action that should be taken next. Typically, there is a value, gamma $\in [0,1]$, which is multiplied against the expected utility of the next action's state's utility. Gamma represents a diminishing return, decreasing utility as it is used. The action is then applied if the racecar ever enters the state, i.

Value Iteration eventually converges to a unique set of solutions (Stuart Russell, 2010). Value Iteration utilized a concept referred to as 'contraction'. Contraction is a function that, when supplied inputs, creates two different outputs that are successively closer together. As a contraction function is called, it continually outputs closer and closer values. A contractive function is a convergent one – that is, it will eventually reach a convergence point in which a similarity is reached. Value Iteration uses contraction as its exiting condition on the outer loop. Because of this, the algorithm does not finish until it has reached a point of convergence.

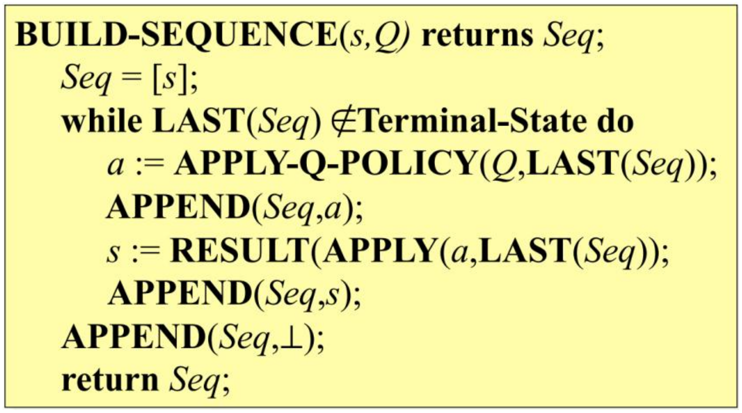
2.2 Q Learning

The second algorithm we look at in this paper is the Q Learning algorithm. As mentioned earlier, the Q Learning algorithm is a model-free algorithm that focuses on a balance of building known states and exploring new states. The major difference between Q Learning and Value Iteration is that Q Learning does not need a transitional model. This is because the Q Learning agent learns the transitional model from applying actions to states. The pseudo-code representing Q Learning is shown below, in Figures 2-4:



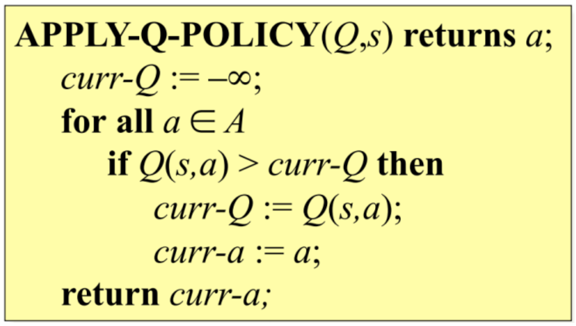
```
UPDATE-Q-VALUES(Seq,Q) returns Q;
    ⟨s,a⟩ = FIRST-PAIR(Seq);
    while $a \neq \bot$ do
        ⟨s',a'⟩ = NEXT-PAIR(Seq);
        if $a' \neq \bot$ then
            $Q(s,a) := (1-\alpha)Q(s,a) +$
                $\alpha(R(s,a) + \gamma \max_{a'} Q(\langle s',a' \rangle))$;
        else
            $Q(s,a) := (1-\alpha)Q(s,a) + \alpha R(s,a)$;
        ⟨s,a⟩ = ⟨s',a'⟩
    return Q;
```

Figure 2 -- Update-Q-Values (Sheppard, 2012)



```
BUILD-SEQUENCE(s,Q) returns Seq;
    Seq = [s];
    while LAST(Seq) $\notin$ Terminal-State do
        a := APPLY-Q-POLICY(Q,LAST(Seq));
        APPEND(Seq,a);
        s := RESULT(APPLY(a,LAST(Seq));
        APPEND(Seq,s);
    APPEND(Seq,$\bot$);
    return Seq;
```

Figure 3-- Build-Sequence (Sheppard, 2012)



```
APPLY-Q-POLICY(Q,s) returns a;
    curr-Q := $-\infty$;
    for all $a \in A$
        if $Q(s,a) >$ curr-Q then
            curr-Q := Q(s,a);
            curr-a := a;
    return curr-a;
```

Figure 4 -- Apply-Q-Policy (Sheppard, 2012)

The Q Learning algorithms comprise of Figures 2-4. Figure 2 shows the process of updating Q values, or the value of performing Action a in State s. Figure 3 shows the process of building a sequence in which a racer will run through. Finally, Figure 4 shows the process of picking the best action at any state.

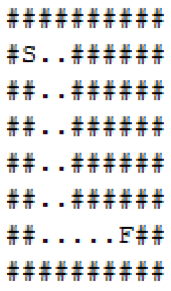
Let us first take a look at Figure 2. This pseudo code is responsible for assigning Q values to a sequence of state action pairs. Q values represent the utility of applying a certain action at a certain state. Each state action pair will have a Q value, which is built up incrementally as the agent extends their sequence by exploring new states. Now, the process of assigning Q values is based on taking a balanced sum of the old Q value and the reward of being in a state along with the value of the highest scoring action from the current state. The variable α is used as the balance between which section more emphasis should be placed on; the old Q value or the updated reward. This function loops through the entire sequence inputted.

Now let us look at Figure 3. This pseudo code is responsible for building a sequence that an agent will run through. Basically, the maximum action, gathered from Figure 4, is appended to the current sequence, followed by the resulting state. This repeats until a terminating condition is found. Quite often it is beneficial to focus on small sequences, first exploring and learning them, then to move on to larger areas. This is a divide and conquer approach to the problem of having an agent hopelessly lost.

Under reasonable conditions, Q Learning converges (LB). Although Q Learning never learns the transitional model, it has similar attributes as Value Iteration. It turns out the transitional model does not matter for convergence, because the frequency of determining a future state depends entirely on the transitional exploration. A future state will never be reached unless the transitional exploration occurs. When that transitional exploration occurs, an optimal path to that state has been found because of the dynamic programming constructs inside Q Learning. Q Learning will therefore, under reasonable conditions, converge.

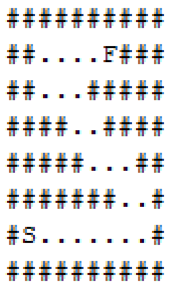
3 Experimental Methods

Numerous experiments were performed using a racer as a Value Iteration learner. The variables of number of movements until the finish state, total running time, varying gamma values, and total collisions were recorded. These variables were all tested with the different penalties for hitting a boundary. Each track was run through 100 times, and the variables gathered were averaged so as to not have randomness influence results. Experiments were performed on the following tracks (see Figures 5 - 8).



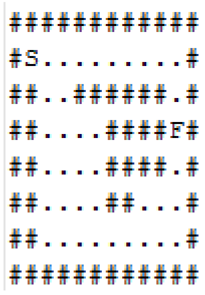
```
##########
#S..######
##..######
##..######
##..######
##..######
##.....F##
##########
```

Figure 5 –
The L track



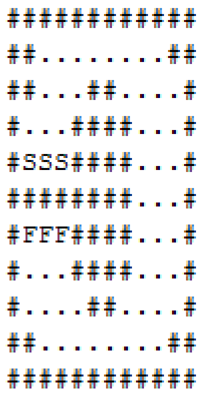
```
##########
##....F###
##...#####
####..####
#####...##
#######..#
#S.......#
##########
```

Figure 6 –
The Z track



```
############
#S.........#
##..######.#
##....####F#
##....####.#
##....##...#
##.........#
############
```

Figure 7 –
The two-way track



```
############
##........##
##...##....#
#...####...#
#SSS####...#
########...#
#FFF####...#
#...####...#
#....##....#
##........##
############
```

Figure 8 –
The O track

Each of these tracks illustrates a different challenge that the racer must overcome. The L track in Figure 5 is a straightforward, maneuverability challenge. The zig-zag track in Figure 6 is a track in which the racer may wish to accelerate slower, to ensure that no walls are collided with. The downside is that a traversal may take more time. The two-way track in Figure 7 illustrates a case where the racer has a choice of paths. If they are feeling risky, the racer can take the shorter, top path. But, if the racer does not want to face the challenge of starting over at start, they can take the longer, more open path. Finally, the O track in Figure 8 shows a track in which the racer must handle acceleration and decelerating in a balanced manner. The racer can risk the inside route, finishing in three moves faster than the outside route, yet may risk losing control of acceleration.

4 Results

The results of running the experiments with Value Iteration are shown in Appendix A. Track L gave the most noteworthy results.

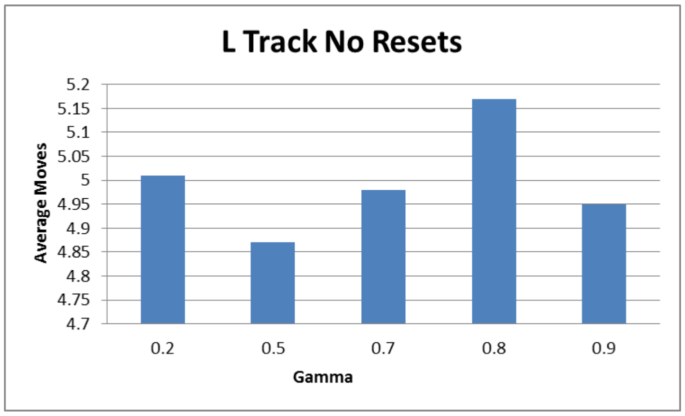


Figure 9 -- Average Moves vs gamma for the L track without resets upon hitting a boundary

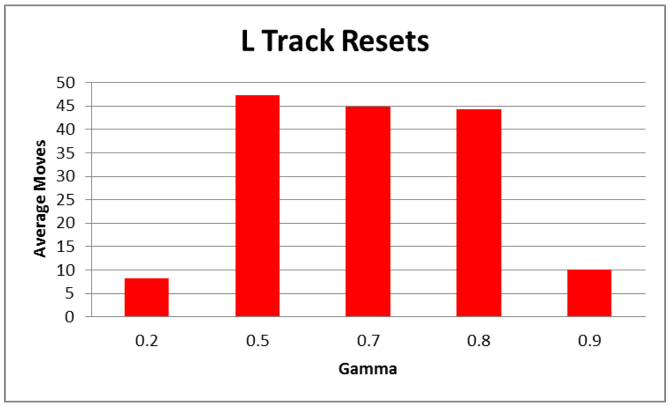


Figure 10 -- Average Moves vs gamma for the L track with resets back to start when hitting a boundary

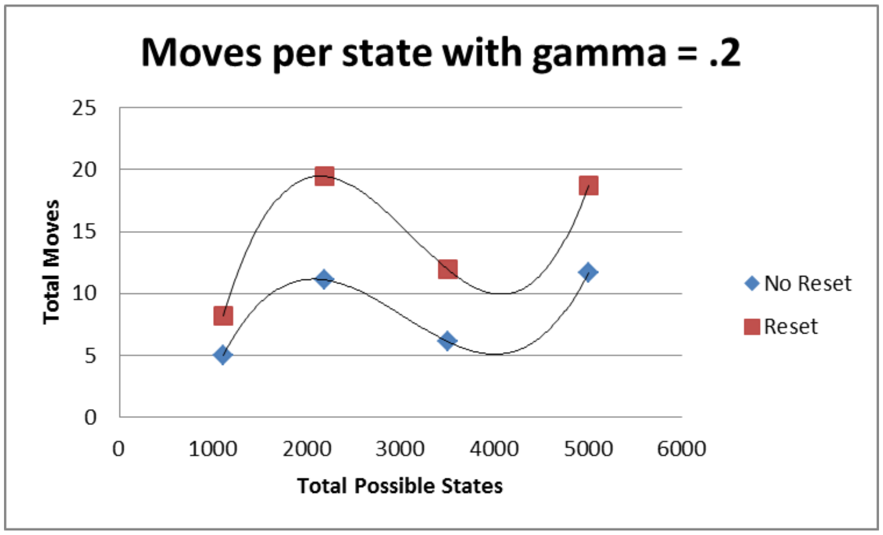


Figure 11 -- Moves per state with gamma = .2

	Average Moves	Average STD	Max moves	Min moves	Collisions (total)	Time(ms)
No-Reset	4.996	1.1632	10	4	56	10295.2
Reset	30.958	25.6246	132.2	5.8	1014	10106.8

Table 1-- Averages for the L track

5 Discussion

Let us first look at the average moves for the L track with the no-reset and reset penalties. Notice that in Figure 9, the average moves follows a spline-like shape. It minimizes at gamma = .5, with a maximum at gamma = .8. This can be attributed to somewhat of an 'optimal' gamma for that track. In other words, we have fine-tuned gamma to find that the quickest determiner for how to reach the finish line fastest is when gamma balances previous and future rewards. Interestingly enough, the gamma values of .9 and .2 fared almost equally. This would be evidence towards equality when favoring either present rewards or future rewards.

In Figure 10, we see almost the opposite. When gamma was balanced, the learner actually took more moves to complete the track. The learner could have had quite the erroneous ride, or simply been very unlucky. In any case, this evidence points towards the fact that it is better to prefer either short term rewards or long term rewards than some combination of them. Maybe the probability of successfully transitioning was too low for the learner, resulting in a dependence on going slower. The difficulty is that the slower a learner moves, the higher a probability the learner will be thrown off course.

Figure 11 shows some validation that the states chosen were accurate at determining difficulty for the learner. The polynomial trend lines are both of power 3, with a very similar equation. This tells us that generally, the learner at gamma = .2 was well represented across resets and no resets. Note that the maximum difference between resets and non-resets is much greater than the minimum difference. One can expect that as more and more states are gathered, resetting learners will take exponentially longer to reach the goal state.

Finally, let us look at Table 1. The results can be expected, for the most part. Note the smaller values in almost every category for the case where the learner does not reset. This learner, on average, used 25 less moves than the resetting learner. The standard deviation is interesting as well; in the case where the learner does not reset, it only varies by around 1 move. Now, because the average number of moves is quite close to the minimum moves taken, and our standard deviation extends beyond the minimum move, we can safely assume our distribution is skewed to the right. This can be expected, because there is an exact minimum number of moves for any state (4 for the L tack), yet there can be a maximum of any number of moves. This leaner is based on transitional probability, so provided enough run-throughs, a maximum move set that exceeds the 10 will eventually occur. Similarly, we can look at what happens when the learner is reset after colliding. This average move count is much large, as well as the standard deviation. Yet, this dataset's curve almost exactly resembles that of the no-reset dataset, with the exception of more spread. It will still be heavily skewed to the right.

6 Conclusions

Ultimately, Value Iteration is a very good reinforcement learning algorithm to use when a model-based environment is available. Because it contains the knowledge of a transitional model, it will (generally) always be faster than Q Learning, which must first learn the transitional model.

For future work, I will look at debugging an implementation of Q Learning, to focus on optimization. Very rarely will a model-free algorithm be able to compete with a model-based one, but such measures are worth recording. The alpha and gamma values associated with Q Learning may need to be fine-tuned in order to define optimal results, which will take numerous experiments.

References

Australia, U. o. (n.d.). *CITS4211 Sequential Decision Problems.* Retrieved December 6, 2012, from University of Western Australia: http://undergraduate.csse.uwa.edu.au/units/CITS4211/Lectures/wk6.pdf

LB, K. (n.d.). *DT5 Lecture.* Retrieved December 6, 2012, from University of British Columbia: http://www.cs.ubc.ca/~kevinlb/teaching/cs322%20-%202007-8/Lectures/DT5.pdf

Sheppard, J. (2012). Q Learning. Bozeman, Montana, United States of America.

Stuart Russell, P. N. (2010). *Artificial Intelligence: A Modern Approach 3rd Edition.* Upper Saddle River, New Jersey: Pearson Education Inc.

Appendix A

Track	Reset	Gamma	States	Average Moves	STD moves	Max moves	Min moves	Collisions (total)	Time(ms)
L	No-Reset	0.2	1105	5.01	1.352	12	4	59	10245
L	No-Reset	0.5	1105	4.87	1.005	9	4	45	10358
L	No-Reset	0.7	1105	4.98	1.159	9	4	54	10255
L	No-Reset	0.8	1105	5.17	1.172	10	4	65	10307
L	No-Reset	0.9	1105	4.95	1.128	10	4	57	10311
L	Reset	0.2	1105	8.22	4.415	23	5	94	9722
L	Reset	0.5	1105	47.26	44.506	203	6	1712	10495
L	Reset	0.7	1105	44.93	38.834	185	6	1596	10311
L	Reset	0.8	1105	44.31	35.381	222	6	1544	10291
L	Reset	0.9	1105	10.07	4.987	28	6	124	9715
zig-zag	No-Reset	0.2	2183	11.1	0.953	14	10	402	49057
zig-zag	No-Reset	0.5	2183	n/a					
zig-zag	No-Reset	0.7	2183	n/a					
zig-zag	No-Reset	0.8	2183	n/a					
zig-zag	No-Reset	0.9	2183	n/a					
zig-zag	Reset	0.2	2183	19.46	10.744	67	11	436	48347
zig-zag	Reset	0.5	2183	n/a					
zig-zag	Reset	0.7	2183	n/a					
zig-zag	Reset	0.8	2183	n/a					
zig-zag	Reset	0.9	2183	n/a					
two-way	No-Reset	0.2	3503	6.1	1.101	11	5	209	132664
two-way	No-Reset	0.5	3503	n/a					
two-way	No-Reset	0.7	3503	n/a					
two-way	No-Reset	0.8	3503	n/a					
two-way	No-Reset	0.9	3503	n/a					
two-way	Reset	0.2	3503	11.93	7.238	41	6	378	125211
two-way	Reset	0.5	3503	n/a					
two-way	Reset	0.7	3503	n/a					
two-way	Reset	0.8	3503	n/a					
two-way	Reset	0.9	3503	n/a					
O	No-Reset	0.2	5002	11.69	1.106	16	10	407	257793
O	No-Reset	0.5	5002	n/a					
O	No-Reset	0.7	5002	n/a					
O	No-Reset	0.8	5002	n/a					
O	No-Reset	0.9	5002	n/a					
O	Reset	0.2	5002	18.73	9.979	89	12	324	261603
O	Reset	0.5	5002	n/a					
O	Reset	0.7	5002	n/a					
O	Reset	0.8	5002	n/a					
O	Reset	0.9	5002	n/a					