# Quagent control via Passive and Active Learning

Computer Science 242: Artificial Intelligence
March 5, 2004


## Rob Van Dam and Greg Briggs



Abstract: Artificial intelligence algorithms using passive and active learning versions of direct utility estimation, adaptive dynamic programming and temporal difference approaches to simulate an agent. The explored worlds consisted of discrete states (positions) bounded by internally generated “walls” that included one or more terminal states and a pre determined configuration of rewards for each state. Passive learning algorithms use a pre-calculated optimal movement policy to travel from the start state to the best terminal state. Active learning algorithms use an initially random movement policy and correct the known policy based on the percepts received within the world. In a passive learning scenario, all three approaches were found to be effective since the optimal policy was known. In active learning scenarios, direct utility estimation tends to result in unsolvable situations or less than optimal policies as a result of poor initial random that create unreachable areas. Adaptive dynamic programming is very effective in an active learning scenario but is inefficient in storage space and often fails to evaluate the entire map. Temporal difference learning approaches are very space efficient but often require more trial runs to approach the same level of accuracy that adaptive dynamic programming achieves. These different learning techniques can be tested with or without the use of quake and the Quagent bot.



Keywords: Quagent, Quake, Active Learning, Passive Learning, AI Exploration, Direct Utility, Adaptive Dynamic Programming, Temporal Difference, Continuous Policy Iteration


Changes in this revision: SenseQuakeWorld and Continuous Policy Iteration added. Much much more analysis.

Potential Bonus credit: For all the analysis.

Note: View and print this document in color.

# Background and Motivation

The Quagent concept is the use of the Quake II 3D video game engine for testing artificial intelligence algorithms. In this paradigm, an agent, or “bot,” interacts with the Quake world under the direction of an external program. We design a learning mechanism to develop a policy for obtaining a desired goal (usually the highest possible reward) within a given world. This world could be represented either essentially internally or through probing a given quake world for information about nearby items and the position of visible walls. We can then test that any given learning mechanism when allowed to iterate successively approaches a known optimal policy or that it successfully fulfills some given goal (such as searching for and gathering all items within a quake world or exploring all unknown areas to effectively create an internal map of the unknown world).

# Methods

Each type of world and each learning algorithm is contained within its own Java class. The main Java Quagent controller is passed parameters to indicate what type of world and learning algorithm to use. The Quagent controller can also be told to use and query an actual Quagent bot within Quake or to just simulate what would happen if the Quagent bot were to perform the same actions. If the Quagent bot is used, the Quagent can then be observed as it moves within the Quake world as constrained by the type of “world” given as a parameter. Simulations work much faster than a Quagent bot but sometimes cannot account for everything that might happen within the Quake program (true Quake worlds are in fact dynamic whereas the simulated worlds as stored are static).

There were three basic learning algorithms that were tested in both active and passive learning scenarios. These were direct utility estimation, adaptive dynamic programming, and temporal difference learning.

## Worlds

A controller was created which allowed for the testing of various learning algorithms in conjunction with moving a quagent bot within different types of quake and non-quake worlds. In the case of the non-quake dependent worlds this algorithms could be tested independently of the actual quake program, allowing for faster run

time.

Tests were performed on a four types of worlds. The first was four by three grid state with one of the states unreachable, one terminal state with negative reward and one terminal state with a positive reward. The second was a ten by ten grid state with a terminal state with a positive reward in one corner. The third was a ten by ten grid with a terminal state with a positive reward in the center. The fourth was an actual quake world as seen by the quagent bot.

**SenseQuakeWorld**

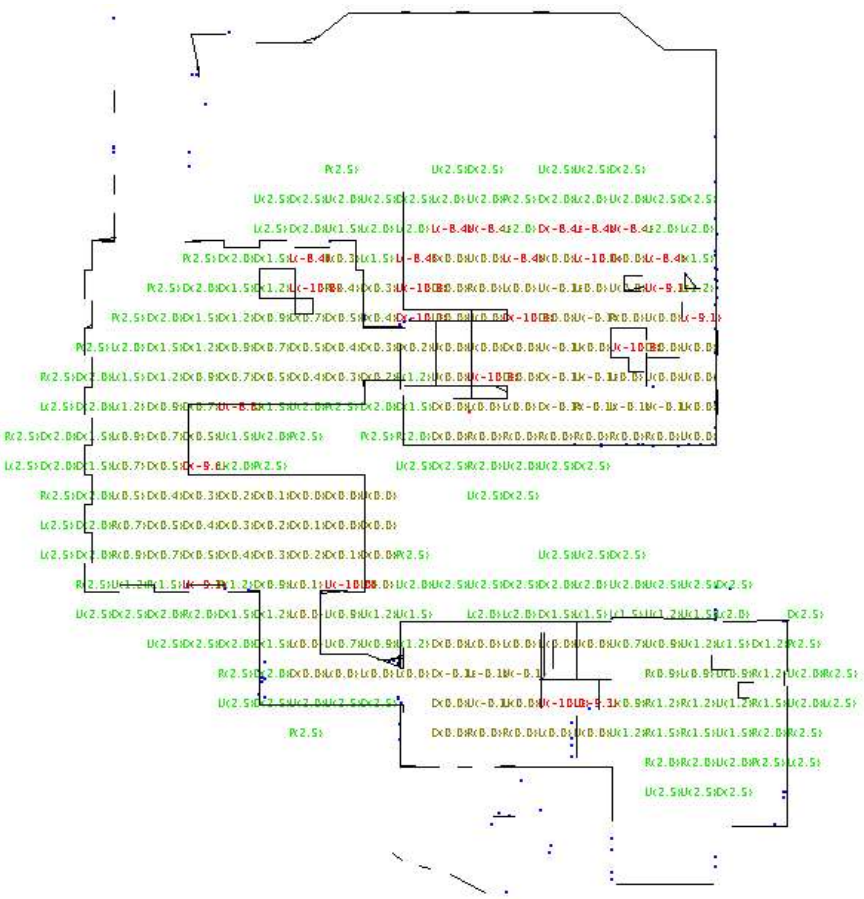
This world was used to allow the bot to use its learning techniques directly in the Quake level. This world was implemented by Greg in the SenseQuakeWorld Java class.

This class essentially learns the MDP from the quagent's sensors, and then presents that MDP to the the MDP problem-solving algorithms. This class presented any useful items (such as Data) as +1 rewards, and harmful items (such as Kryptonite) as -1 rewards.

To learn the MDP, states near the current location, spaced 65 units apart, are generated. The mapping ability was used to identify valid transitions.

Occasionally, certain states were reachable according to the map, but not reachable in reality. Often, these are cliffs. These unreachable states are noted by SenseQuakeWorld and assigned negative rewards.

To encourage exploration, states on the “fringe” of the currently known states were assigned an optimistic 0.5 reward.



Above, the quagent is learning the map and the ideal policy. Note that the color of the policy denotes the utility, ranging from green for positive to red for negative. (The exact color was computed using the sigmoid function, which can map an infinite range of utilities into a finite range of colors.) Notice the green exploration fringe, as well as the red cliff in the top-center room.

It was decide to put the task of learning the MDP into the SenseQuakeWorld, so that algorithms such as Continuous Policy Iteration could be used. Additionally, the transitions are learned much more quickly here than in a generic transition-learning algorithm, because the agent's sensors can be directly exploited. Of course, transition and MDP-learning algorithms can still be attached to this world definition and simply ignore the extra information.

## Learning Algorithms

Several learning algorithms were implemented, most in both a passive as well as an active form. These are Direct Utility Estimation, Adaptive Dynamic Programming (ADP), Temporal Difference (TD), and Continuous Policy Iteration.

Three algorithms (DU, ADP, and TD) were tested in passive learning scenarios. This consists simply of giving the algorithm initially the optimal policy as calculated from the known states and then comparing the utilities that it assigns to the world as it follows this optimal policy to the optimal utilities expected. Generally all three algorithms approach the initially calculated values of the utilities for the optimal policy.

The algorithms were also tested under an active learning scenario. In this type of scenario, each algorithm is given an initial random policy from which it attempts to discover the optimal policy iteratively as in adjusts the utilities of each state over time depending on the rewards it receives while following the current policy.

**Direct Utility Estimation**

The Direct Utility Estimation algorithm was written initially by Greg and later modified by Rob. This algorithm tries to use the information gathered from traversing the states of the world to recalculate the appropriate utilities. No change is made to the known utilities until the quagent has reached a terminal state.

The passive version of this algorithm only adjusts the utilities, which under ideal circumstances will tend to approach very closely the utilities as calculated from the optimal policy.

The active version of this algorithm also adjusts the policy itself to account for the rewards received within the world. This is done specifically by giving the quagent an initially random policy to follow and then allowing it to follow this policy and learn from the results.

**Adaptive Dynamic Progamming**

The Adaptive Dynamic Progamming algorithm was written by Rob. This algorithms tries to use the percepts received from the world to adjust the known transition model to learn the transition model needed to recreate the optimal policy. This is a much different approach to learning how to maneuver in the world than is Direct Utility Estimation.

The passive version only attempts to learn the Transition model based on the frequency with which performing a given action in a given state results in another specific state.

The active version also adjusts the policy through the percepts that it

receives and attempts to learn the optimal policy and transition model. It uses modified policy iteration using value iteration to update the utilities as suggested by Russell and Norvig (AI, p. 767).

### Temporal Difference

The Temporal Difference algorithm was written by Rob. The temporal difference algorithm uses a table of frequencies similar to the ADP algorithm but it does not calculate the transitions. Instead it uses a specific learning algorithm to recalculate the utilities of the world based on rewards received there. The frequency table is used to determine how pertinent the percept is to the new calculation. That is to say that the new utility of each state visited is recalculated using the old utility, the reward received at that state and the utility of the next state reached. A learning rate function is necessary for this algorithm to work. The learning rate function determines how heavily the algorithm relies on current knowledge versus new percepts. The algorithm as described by Russell and Norvig (AI, p. 769) is:

$U[s] = U[s] + \alpha(N_s[s])(r + \gamma U[s'] - U[s])$
s is previous state
r is previous reward
s' is current state
$\alpha$ is the learning rate function
$\gamma$ is the discount factor

The passive version only modifies the utilities. The active version also modifies the policy instantaneously based on each percept, which will impact subsequent knowledge gain.

### Continuous Policy Iteration

This algorithm, written by Greg, was aimed primary at the ever-changing Quake world. As new states are added to the MDP, the policy is updated with random actions for the new states, and the new utilities are assigned zero. Next, policy iteration is executing using this policy and utility set, which calculates meaning values for these. This algorithm takes advantage of a gamma (that is, learning rate or discount factor) less than one in order to simplify computations as the number of states grows very large.

## Standardized Interface For Worlds And Solution Algorithms

Taking advantage of Java's ability to specify interfaces, Greg created

an interface for a Markov Decision Problem definition (the world), and for a Learning Algorithm that will guide an agent in solving such a problem. Depending on the type of learning algorithm, more or less of the information from the MDP can be used. For example, although the MDP definition must specify a transition model and set of states, the learning algorithm can still attempt to learn it's own model and state set.

### Dynamic MDPs

To support representation of the Quake world as an MDP, it was necessary for Greg to allow new states to appear in the MDP while the agent is executing a learning algorithm. For the algorithms that were not intended to support this, it was necessary to modify them. Algorithms which maintain a policy were modified to initialize the new states with random actions. Algorithms which assess utilities were modified by giving a default utility value to these new states. Not all algorithms were modified to support a changing MDP.

### Modified Policy Iteration

Several algorithms needed to compute the “ideal” utility values. The policy iteration algorithm that Greg coded used iteration to solve for the utilities of the states. This iteration stopped when the values stopped changing by more than a constant, or when a maximum count had been reached.

### Learning by Many Trials

For the algorithms that needed to be run over and over, the agent would be restarted in a random state in the MDP, rather than the original start state.

# Results

## Policy Iteration

Verification of Utility values determined by Policy Iteration in the Four Three World [G]

Policy Iteration Result

| | | | |
|---|---|---|---|
| 0.812 | 0.868 | 0.918 | 1 |
| 0.762 | XX | 0.660 | - 1 |

Twenty Trial Average

| | | | |
|---|---|---|---|
| 0.81 | 0.87 | 0.92 | (1) |
| 0.76 | XX | 0.66 | (- 1) |

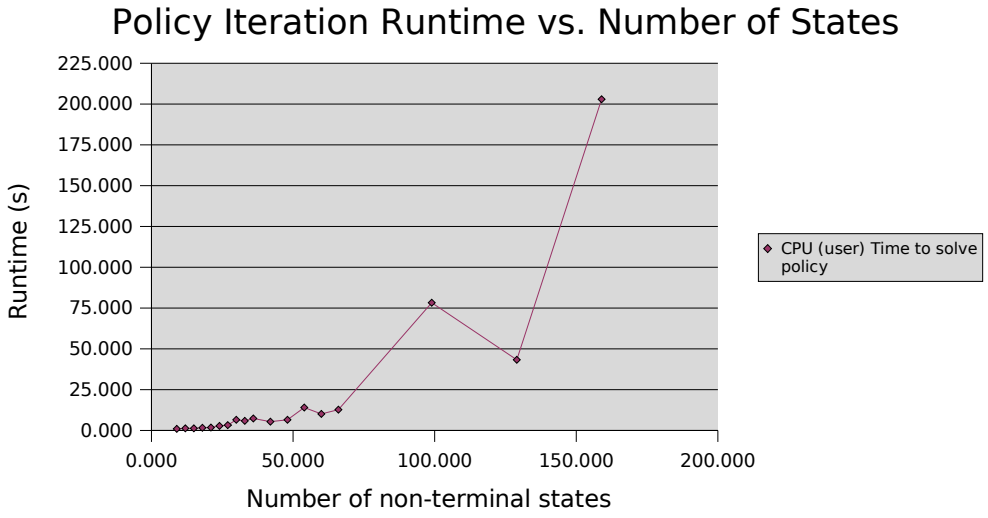
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0.705 | 0.655 | 0.611 | 0.388 | | 0.71 | 0.55 | 0.54 | 0.48 |

The policy iteration result has an RMS difference from the twenty trial average of .052.

Note that the policy iteration result exactly matches the result found by Russell and Norvig (AI, p. 619).

### Policy Iteration Runtime

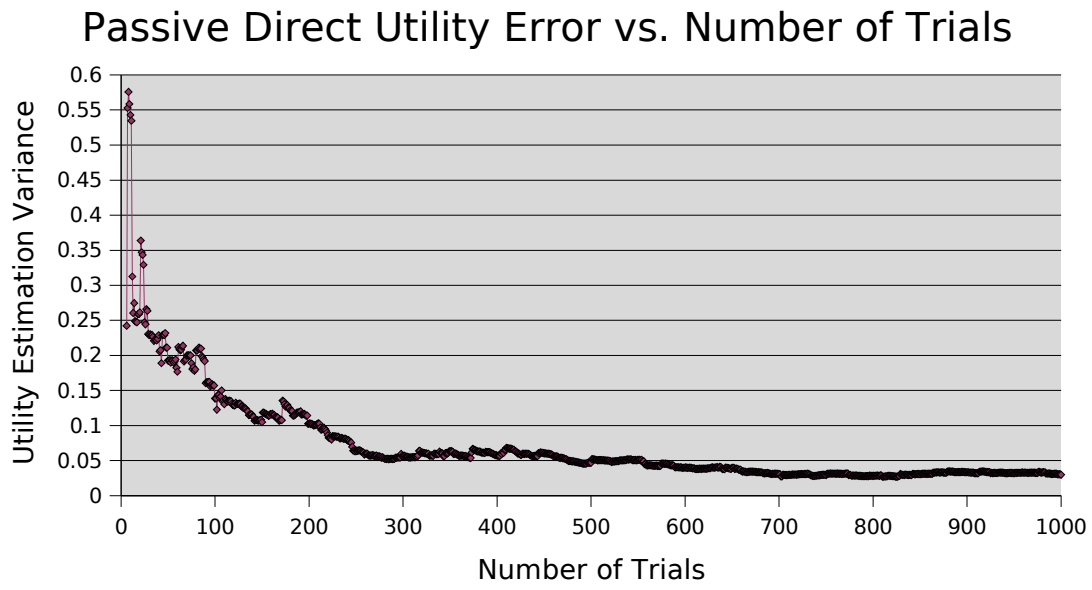
As the runtime is important for the feasibility of continuous policy iteration, it is considered here. The below graph depicts an gamma of 1. By using a lower discount factor, this growth could be inhibited.

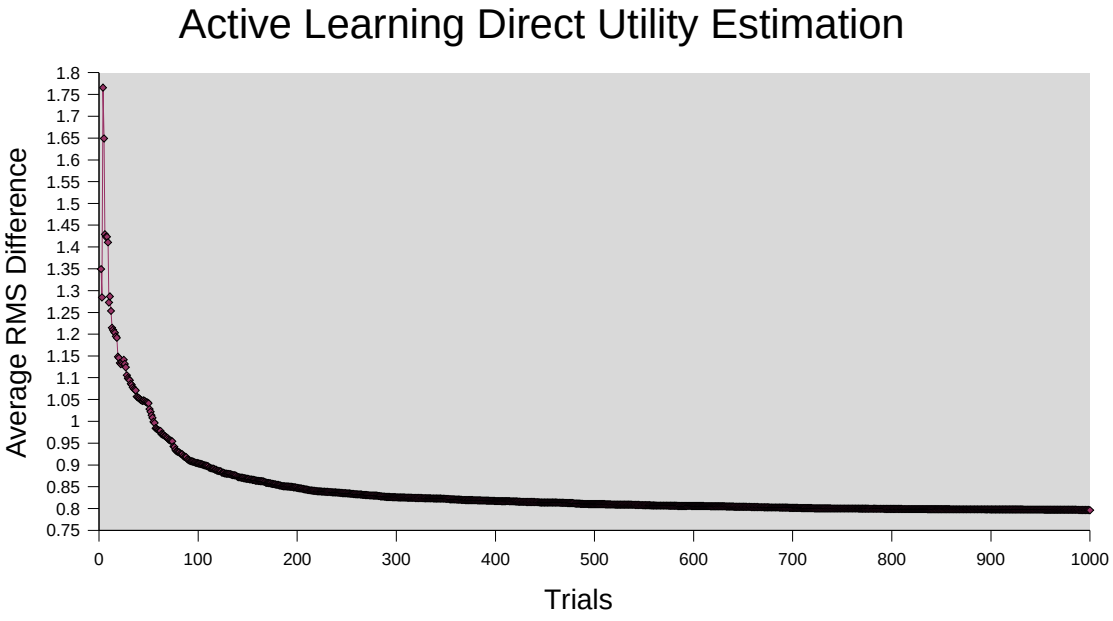



## Direct Utility Learning

As shown below, the error seems to be related to the number of trials by an exponential decay. The “bumps” represent sudden significant information, which was falling into the -1 state in this case. Note that this is the result of 5 runs averaged together; each indidual run has larger variations.

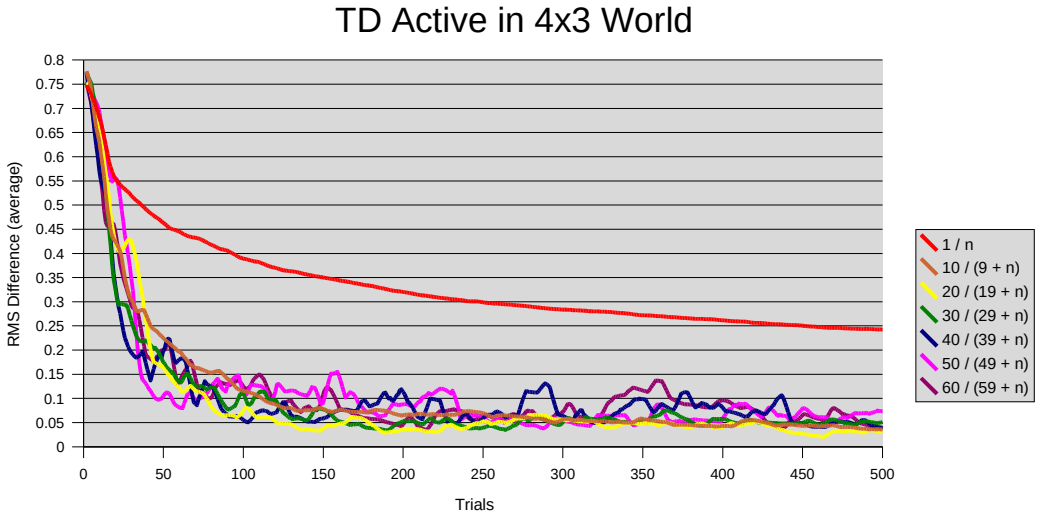
Below, we see the result of Active Learning DU. One will notice from the scale of the graph that it was about 3 times slower in learning. In this particular case, rather than random restarts, both this active and the passive DU we restarted in the same spot every time. This allowed the active DU to sometimes “rule off” part of the world and then never revisit it to update its estimations, leaving erroneous utilities.


Passive Direct Utility Error vs. Number of Trials
Utility Estimation Variance
0.6
0.55
0.5
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
0
100
200
300
400
500
600
700
800
900
1000
Number of Trials


Active Learning Direct Utility Estimation
Average RMS Difference
1.8
1.75
1.7
1.65
1.6
1.55
1.5
1.45
1.4
1.35
1.3
1.25
1.2
1.15
1.1
1.05
1
0.95
0.9
0.85
0.8
0.75
0
100
200
300
400
500
600
700
800
900
1000
Trials

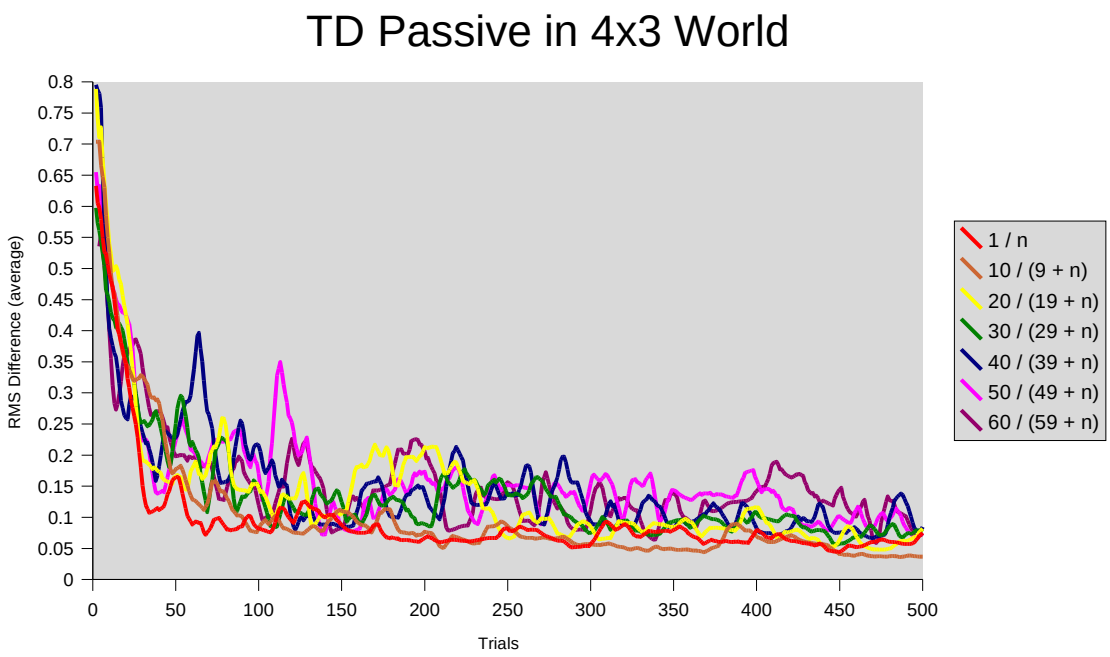
## Temporal Difference Learning

Below, we see the result of using various alpha functions in both passive and active learning. The alpha function is what TD uses to weigh new knowledge against previous knowledge, where “n” is the number of items already known. For example, the function “1/n” puts little weight in new knowledge, whereas “60/(59+n)” (as suggested by Russell and Norvig) puts significant weight on new knowledge.

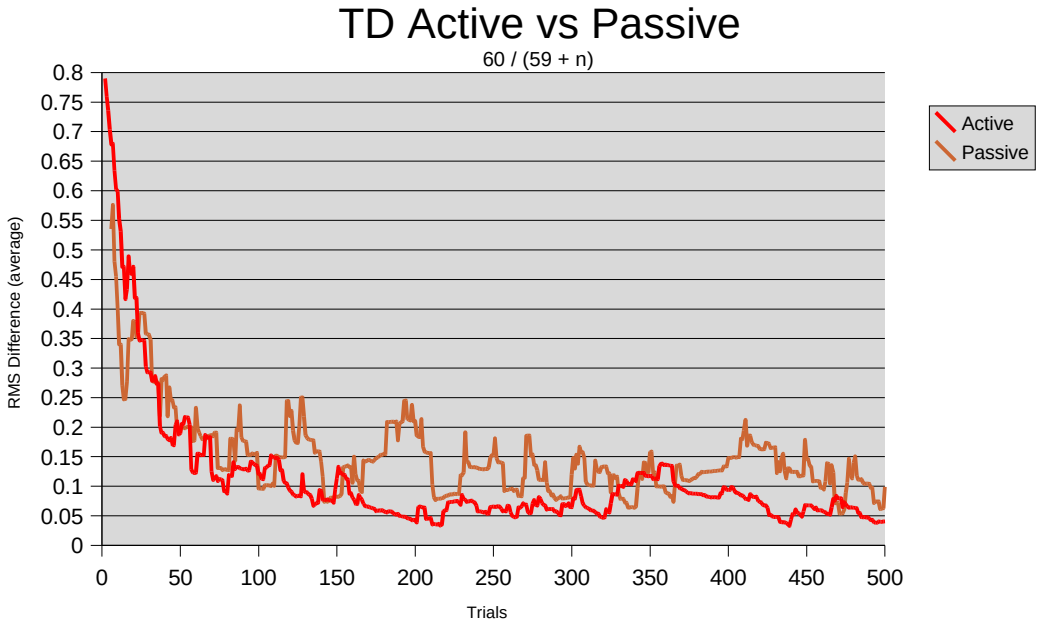



In the above graph, everything except 1/n seems to do well. The explanation for this is that, during the active learning process, the agent will tend to practically ignore new discoveries due to their low weight. The other alpha functions all these new discoveries to be investigated. Also note the 1/n line is smoother because new information is not able to change things significantly.

In the passive mode, below, the “1/n” actually does best, because using this alpha function calculates an accurate running average, weighting each piece of information, new or old, equally. Since the policy is fixed in advanced, this does not hinder exploration as it did in active mode.

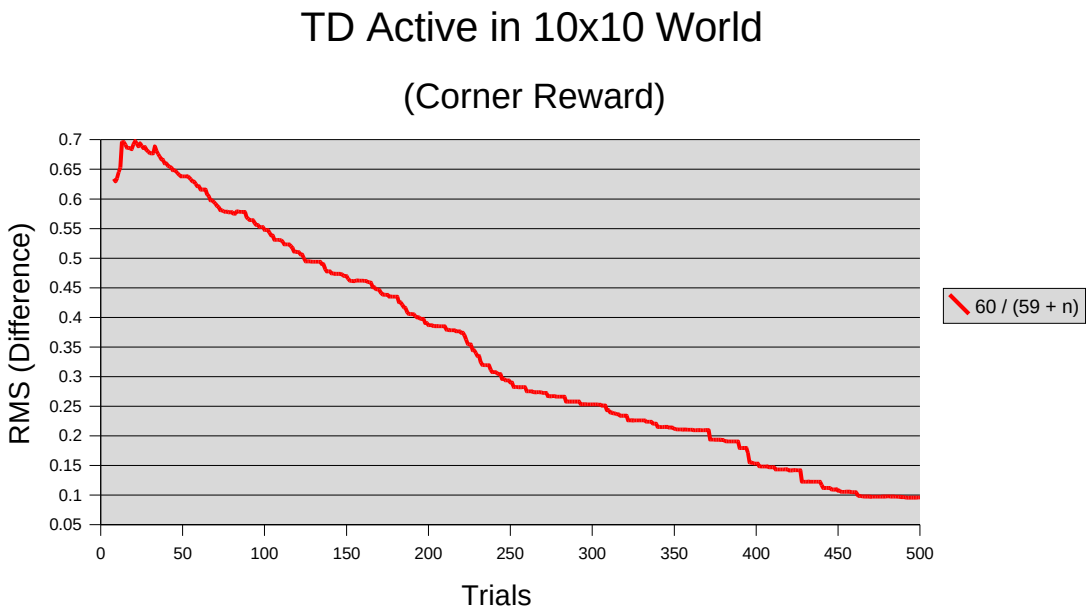



Below, we see that active learning actually proceeds faster than passive learning, although this may be less true for different alpha functions.




Finally, in comparing the 10x10 world below with the above graphs, one notices that the convergence happens much more slowly, in an almost linear fashion, or rather having a slower exponential decay. This appears to be connected to the amount of time the agent is

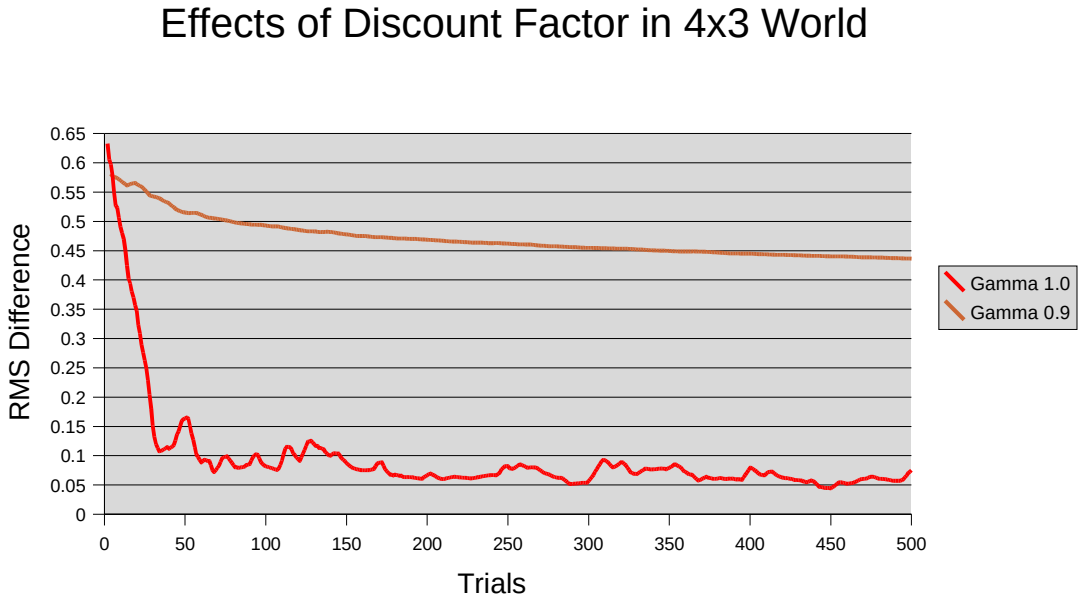
spending learning relatively new information (since there are more states to learn about), rather than spending time refining that which it




already knows.

## Discount Factor

One of the factors that greatly impacts the effectiveness of any of the learning algorithms is the discount factor. Altering this factor from the default of 1.0 can be very effective for some types of algorithms and prove very detrimental for other algorithms. In the case of the TD algorithm as employed in the 4x3 world or any similarly pre-known world (such as a 10x10 world) a lower discount factor is very detrimental. This is because the factor basically defines the algorithms responsiveness to new information. If it is close to one, it favors new information, allowing for quick changes to the known model. This in turn tends to favor exploration of unknown states. If it is lower, it favors current knowledge in an effort to avoid possibly “dangerous” outcomes.




In a small, simple world like the 4x3 world, exploration is not necessary after only a few trials and it is more important to adjust the knowledge base with new information. As can be seen in the above graph, an algorithm with even a slightly lowered discount factor is slow to learn. A subtle benefit to this is there are fewer suddent changes. In an unknown world such as the Quake World this is beneficial. It is more important to explore, as unexplored areas may contain something of interest that would not be found otherwise so it is effective to lower the discount factor slightly to allow for this.

# Discussion

## Learning Algorithms

### Direct Utility Estimation

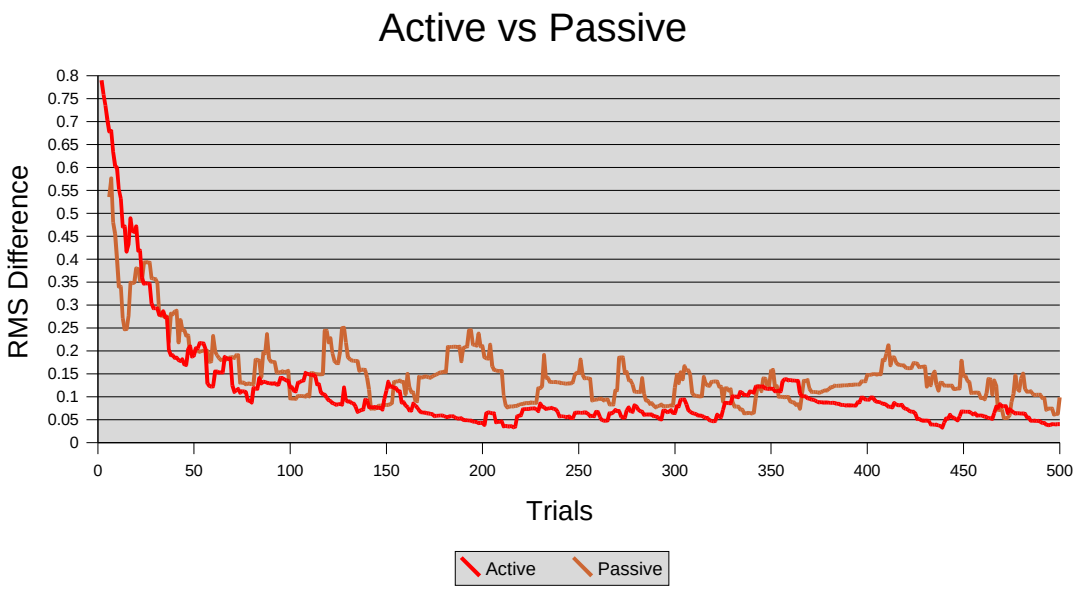
Direct Utility Estimation tends to be fairly quick in its calculations however the active version has some shortcomings. First, if the initial random policy prohibits the quagent from accessing some part of the world, it cannot learn the optimal policy for that section of the world. Second, since the utilities and policy are calculated only once a terminal state has been found, a world with no pre-known or pre-defined terminal state would not allow the quagent to learn from its actions, nor have a basis for what actions to take. The initial shortcoming can be resolved by allowing the quagent to start each iteration in a randomly selected state, thereby eliminating unreachable states due to a poor initial policy. The second

shortcoming can only be resolved by using a different learning algorithm.

**Adaptive Dynamic Programming**

The Adaptive Dynamic Programming solution runs very quickly and is relatively simple in design and code. It however requires large amounts of memory to store the frequencies with which one state is reached from another given some action. In a small world such as the 4x3 world this is not prohibitive. However, in a large world this would require very large arrays of information. In a world of unknown size, this would require the use of expandable arrays or some other data structure that could grow with time. This too could in turn be expensive at runtime. ADP also has the shortcoming that it has to solve all of the equations using the transition model that it calculates in each iteration. Again, for a small world this is not very difficult but in a large world with many states this could be very slow at runtime.

For the Temporal Difference algorithm, the passive version only modifies the utilities, whereas the active version also updates the policy as new information is discovered. This allows for much faster reaction to new knowledge. The dependence on current knowledge is defined heavily by the learning rate function. Since the active version starts with an initially random policy, it too has the possible shortcoming of creating unreachable sections of the world. However, by starting the quagent in a random state in each iteration, this shortcoming is eliminated and the data received is much closer to the optimal because the quagent gets the chance to observe the entire world more equally.

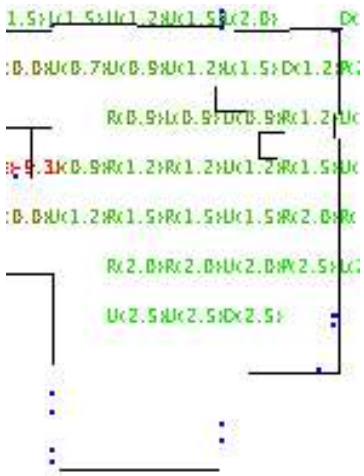



The passive version of the algorithm exhibits a higher degree of "jitter" in the RMS difference because it does not adjust its policy to the known utilities. Since the active version does, it is much less likely to uncover situations that cause it to have to make drastic changes in the utilities at any given point.

Because of the efficient runtime and storage of the TD algorithm, it was used as a primary test algorithm for making several comparisons. Since its performance can depend greatly on the parameters of the above equation, it is necessary to find which parameters provide optimal results.

**Continuous Policy Iteration**

Learning factors (gammas) of 1.0, 0.9, and 0.8 were tried. The higher numbers provide more "foresight," but the lower number allow calculations to converge more quickly. One concrete impact of this was that, in the Quake world, 0.9 or 1.0 would cause the agent to explore the bottom of the room where it starts, but at 0.8 it would move on right to the next room without finishing the first, as depicted below. Notice, in the image below, how the agent did not get close enough to the bottom to even generate all the states for it. (Also, be aware that the directional letters shown on the map "up" "down" "left" and "right" are oriented such that up in to your right. Now that the bot has already left the room, it is no longer useful to leave the room, so if it were to return, it would be directed to explore the rest of it.)



## Common Algorithmic Issues

### FourThreeWorld versus QuakeWorld

Algorithm behavior in QuakeWorld was less effective. The robot tended to become very sluggish with calculations. Additionally, the utility values converged much more slowly. Especially when starting with a random policy, the agent would get stuck. Due to the real-time nature of the actual Quake world, this took too long.

### Time to evaluate utilities in the Quake World

One algorithmic issue discovered was that, since the current position is almost never precisely one of the predefined states, one needs to find the nearest predefined state that matches reality. This was implemented by checking all states to see the closest. As a result, in a few places (particularly, in determining the utilities from a policy) an extra loop, increasing the order of the runtime, was introduced. To improve efficiency, it would be useful to be able to directly access nearby states using a hash-based data structure. This is certainly a possibility for future implementations.

### Modified Policy Iteration

Since the utility evaluation was inexact, we found that it was in fact possible to get into a infinite loop between two policies. To address this issue, the policy iteration loop saves the past 2 policies, and verifies that the new policy is different from them, to detect and exit such loops. Unfortunately, with gamma values that are equal or extremely close to 1, it is also possible to get into infinite loops containing more than two steps, in which case the algorithm may run

forever.

## Concluding Remarks

Several algorithms were examined, and the details of their behavior were discovered and analyzed in-depth. Additionally, these algorithms have shown themselves as effective solutions to problems such as the Quagent paradigm.

## References

Many of the algorithms were based on Artificial Intelligence: A Modern Approach by Stuart Russell and Peter Norvig, 2nd Ed., 2003.

Some of the viewer code (DisplayGraph.java) was based on code by Michael Scott, dated November 2001.

The Java source code used for the Quagent was based on the DrunkenMaster sample code from CSC 242 class, Spring 2004, University of Rochester.