# CS 5313/7313: Advanced Artificial Intelligence: Solving Markov Decision Processes and Multi-armed Bandit Problems

Noah Schrick, Noah Hall, Jordan White

November 2021

## Contents

# 1 Introduction

The primary focus of this project was in solving Markov Decision Processes and Multi-armed Bandit problems. Within the Markov Decision Process portion of the project, we were tasked with two types of MDPs: those with the world model available, and those with the world model unknown. For the case where the world model is available, we implemented value iteration and policy iteration for various domains, and were to evaluate efficiencies and scale-up properties. For the world model unknown case, we implemented Q-Learning and SARSA for various domains, and were to also evaluate efficiencies and scale-up properties. Lastly, for the Multi-armed Bandit problem, we implemented a UCB algorithm and an $\epsilon$-greedy algorithm, where we were to evaluate the effect of decision accuracy on varying the number of actions and the number of samples.

# 2 World Model Available

## 2.1 Algorithms

In Part 1 of the project, we were tasked to solve various MDPs when we had access to the world model. Namely, we were to implement value iteration and policy iteration according to the implementation specified by Chapter 17 of the textbook. Using these algorithms, we would solve Markov Decision Processes for Grid-World, the Wumpus-World, and Cartpole.

### 2.1.1 Value Iteration

The premise of value iteration is that, for an MDP with a finite number of states, the utility of each state will be calculated. To accomplish this, the algorithm performs many iteration steps in which a Bellman update is calculated for each state. What this does is to calculate the utility value of each state by using the utility values of all that state's neighbors from the prior iteration step. The states are eventually ordered correctly from greatest to least utility values, even if the magnitudes are not exactly correct.

### 2.1.2 Policy Iteration

For policy iteration, the policy of the MDP is initialized randomly, and then the algorithm switches between policy evaluation and policy improvement. In policy evaluation, the utility of each state is calculated if the current policy were run. In policy improvement, a new policy is calculated using one step look-ahead based on the values found in policy evaluation. Eventually, when the policy improvement does not yield a change to the policy, the algorithm terminates, and the policy is guaranteed to be optimal.

## 2.2 Domains

### 2.2.1 Grid-world

Grid-world is a MxN grid in which an agent must make choices to move from one state to another and try to maximize it's utility. In grid-world, there are goal states (with standard utility = 10) and pit states (with utility = -10). Each

action is stochastic in that there is a chance the agent ends up in a different square from what it intended.

#### 2.2.2 Wumpus-world

Wumpus-world is an MxN grid, like grid-world, in which the agent tries to maximize its utility. States that harm the agent's utility are those with a Wumpus and those with a pit. Some implementations of wumpus-world are imperfect information environments in which the agent must rely on sensors to make inferences about states, but we implemented wumpus-world as a perfect information environment.

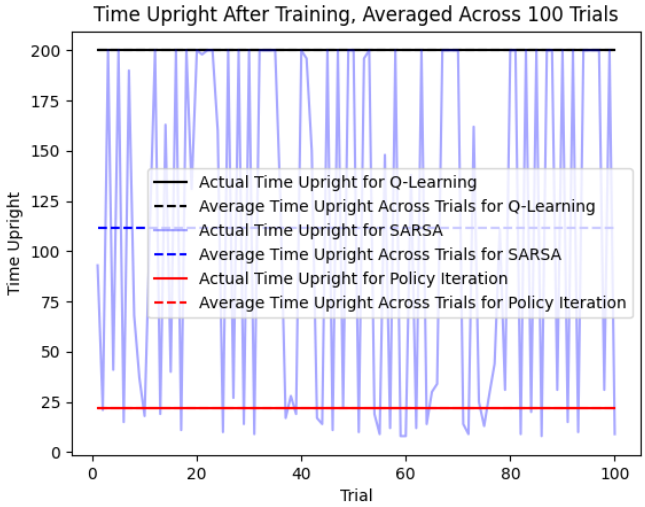
There is also gold located in some states, which provides some positive utility, which the agent may freely traverse to and pick up if he deems it to be worth the living-cost.

#### 2.2.3 Cartpole

For the Cartpole domain, we utilized the OpenAI gym cart-pole environment. In addition, regarding implementation, we utilized the Policy Iteration solution that interfaces with the cart-pole environment provided by OpenAI. This code can be located at https://github.com/vsindato/cartpole-balancing , and it is included as a git submodule in our submitted project code. To allow for comparable results with the Q-Learning and SARSA solutions using the same OpenAI gym cart-pole environment, we kept the parameters consistent. We utilized 50 policy iterations, and 1000 episodes, and measured the rewards per episode, as well as the time upright after training (finding the optimal policy) was complete. In regards to episode length, we set our goal to remain upright for 200 time steps.

## 2.3 Analysis

### 2.3.1 Cartpole




For analysis of this specific domain for this implementation, it worked very well in terms of achieving the goal of staying upright for 200 time steps. After the optimal policy was achieved, remaining upright for 200 time steps seemed relatively guaranteed, and there were no cases where the cart-pole fell or moved off-screen before 200 time steps elapsed. However, there are some downsides. In terms of state-space, this domain had to account for both pole angle and cart velocity. Coupled with a time-frame of 200 time steps and an additional constraint of staying on-screen, runtime complexity was substantially higher, even with "only" 50 policy iterations. So while it did achieve its goal, it came at the cost of a higher runtime complexity. Comparisons to Q-Learning and SARSA will be discussed further in later sections of this report.

As a note regarding the resulting image, the Policy Iteration upright time appears to be flat at a value of 20. The resulting data is correct and is upright for the full 200 timesteps, but the environment is off by a factor of 10 when plotting the results, making the resulting image appear as 20 instead of 200. The environment has since been adjusted, but due to the lengthy runtime of the program, new data was not collected.

### 2.3.2 Grid-world and wumpus-world

Below is example output for the value and policy iteration.

```
6985007638128, (8, 0): -0.1427530000000002, (8, 1): -0.1427530000000002, (8, 2): -10, (8, 3): -0.1427530000000002, (8, 4): -
0.1367718340855, (8, 5): -0.107612422591, (8, 6): 0.011430364125875026, (8, 7): 0.48936191700875, (8, 8): 2.3718074663053437, (
8, 9): 9.608993835062313, (9, 0): -0.1427530000000002, (9, 1): -0.1427530000000002, (9, 2): -0.1427530000000002, (9, 3): -0.
1398655540593, (9, 4): -0.12178454478249999, (9, 5): -0.03712887787899999, (9, 6): 0.3410733002720001, (9, 7): 2.030698500763812
8, (9, 8): 9.608993835062313, (9, 9): 10}
```

Figure 1: Example output for value iteration, showing the utility values for each state.

```
pi: {(0, 0): <Actions.DOWN: 2>, (0, 1): <Actions.RIGHT: 4>, (0, 2): <Actions.UP: 1>, (0, 3): <Actions.RIGHT: 4>, (0, 4): <Actio
ns.RIGHT: 4>, (0, 5): <Actions.RIGHT: 4>, (0, 6): <Actions.LEFT: 3>, (0, 7): <Actions.UP: 1>, (0, 8): <Actions.UP: 1>, (0, 9):
<Actions.RIGHT: 4>, (1, 0): <Actions.RIGHT: 4>, (1, 1): <Actions.RIGHT: 4>, (1, 2): <Actions.RIGHT: 4>, (1, 3): <Actions.RIGHT:
 4>, (1, 4): <Actions.RIGHT: 4>, (1, 5): <Actions.DOWN: 2>, (1, 6): <Actions.RIGHT: 4>, (1, 7): <Actions.UP: 1>, (1, 8): <Actio
ns.LEFT: 3>, (1, 9): <Actions.RIGHT: 4>, (2, 0): <Actions.RIGHT: 4>, (2, 1): <Actions.RIGHT: 4>, (2, 2): <Actions.RIGHT: 4>, (2
, 3): <Actions.RIGHT: 4>, (2, 4): <Actions.UP: 1>, (2, 5): <Actions.UP: 1>, (2, 6): <Actions.RIGHT: 4>, (2, 7): <Actions.DOWN:
2>, (2, 8): <Actions.RIGHT: 4>, (2, 9): <Actions.UP: 1>, (3, 0): <Actions.RIGHT: 4>, (3, 1): <Actions.RIGHT: 4>, (3, 2): <Actio
ns.RIGHT: 4>, (3, 3): <Actions.RIGHT: 4>, (3, 4): <Actions.RIGHT: 4>, (3, 5): <Actions.LEFT: 3>, (3, 6): <Actions.UP: 1>, (3, 7
): <Actions.RIGHT: 4>, (3, 8): <Actions.RIGHT: 4>, (3, 9): <Actions.RIGHT: 4>, (4, 0): <Actions.RIGHT: 4>, (4, 1): <Actions.UP:
 1>, (4, 2): <Actions.RIGHT: 4>, (4, 3): <Actions.RIGHT: 4>, (4, 4): <Actions.DOWN: 2>, (4, 5): <Actions.RIGHT: 4>, (4, 6): <Ac
tions.UP: 1>, (4, 7): <Actions.RIGHT: 4>, (4, 8): <Actions.RIGHT: 4>, (4, 9): <Actions.RIGHT: 4>, (5, 0): <Actions.UP: 1>, (5,
1): <Actions.UP: 1>, (5, 2): <Actions.UP: 1>, (5, 3): <Actions.RIGHT: 4>, (5, 4): <Actions.RIGHT: 4>, (5, 5): <Actions.RIGHT: 4
>, (5, 6): <Actions.RIGHT: 4>, (5, 7): <Actions.RIGHT: 4>, (5, 8): <Actions.RIGHT: 4>, (5, 9): <Actions.RIGHT: 4>, (6, 0): <Act
ions.UP: 1>, (6, 1): <Actions.UP: 1>, (6, 2): <Actions.UP: 1>, (6, 3): <Actions.UP: 1>, (6, 4): <Actions.UP: 1>, (6, 5): <Actio
ns.UP: 1>, (6, 6): <Actions.RIGHT: 4>, (6, 7): <Actions.RIGHT: 4>, (6, 8): <Actions.RIGHT: 4>, (6, 9): <Actions.RIGHT: 4>, (7,
0): <Actions.UP: 1>, (7, 1): <Actions.UP: 1>, (7, 2): <Actions.LEFT: 3>, (7, 3): <Actions.UP: 1>, (7, 4): <Actions.UP: 1>, (7,
5): <Actions.UP: 1>, (7, 6): <Actions.UP: 1>, (7, 7): <Actions.RIGHT: 4>, (7, 8): <Actions.RIGHT: 4>, (7, 9): <Actions.RIGHT: 4
>, (8, 0): <Actions.RIGHT: 4>, (8, 1): <Actions.DOWN: 2>, (8, 2): <Actions.UP: 1>, (8, 3): <Actions.UP: 1>, (8, 4): <Actions.UP
: 1>, (8, 5): <Actions.UP: 1>, (8, 6): <Actions.UP: 1>, (8, 7): <Actions.UP: 1>, (8, 8): <Actions.RIGHT: 4>, (8, 9): <Actions.R
IGHT: 4>, (9, 0): <Actions.UP: 1>, (9, 1): <Actions.UP: 1>, (9, 2): <Actions.RIGHT: 4>, (9, 3): <Actions.UP: 1>, (9, 4): <Actio
ns.UP: 1>, (9, 5): <Actions.UP: 1>, (9, 6): <Actions.UP: 1>, (9, 7): <Actions.UP: 1>, (9, 8): <Actions.UP: 1>, (9, 9): <Actions
.RIGHT: 4>}
```

Figure 2: Example output for policy iteration, showing the best action for each state.

Shown below are the performance times of value iteration and policy iteration on both grid-world and wumpus-world.

```
[0.03604240000000003, 0.0460603, 0.0540988, 0.062024199999999974, 0.08235380000000003, 0.15477129999999995, 0.13282820000000006]
```

Figure 3: Time taken, in seconds, to run value iteration on grid-worlds of size 4x4 up to 10x10.

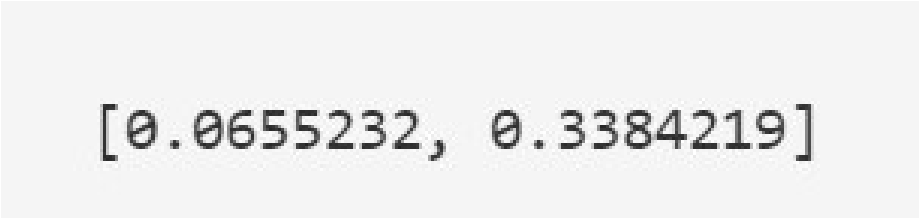



Figure 4: Average time taken, in seconds, to run value iteration on Wumpus-worlds of size 3x4 and 8x10, respectively.

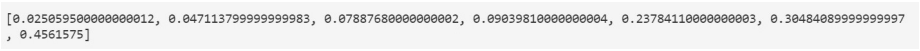



Figure 5: Time taken, in seconds, to run policy iteration on grid-worlds of size 4x4 up to 10x10.

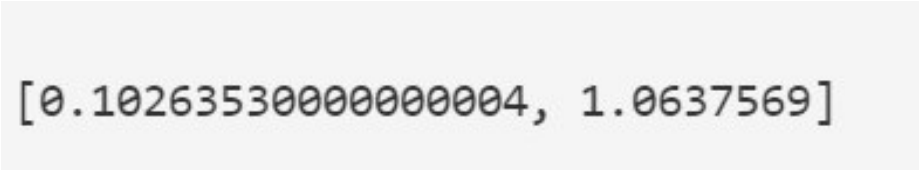



Figure 6: Average time taken, in seconds, to run policy iteration on Wumpus-worlds of size 3x4 and 8x10, respectively.

As you can see from figures 3 and 5, value iteration is significantly faster than policy iteration for exactly the same MDPs.

As you can see in figures 4 and 6, policy iteration also takes much longer to run than value iteration for Wumpus-world.

This makes sense, because policy iteration also calculates the utility values for each state like value iteration, but adds a policy improvement step on top of that.

# 3 World Model Unknown

## 3.1 Algorithms

The two algorithms implemented in this section are Q Learning and SARSA. Something important to note is this implementation uses a modified version of the supplied gridworld.py file. The modifications to the gridworld.py file include adding the parameter 'goal', and changing the def r function to only require

one variable. In order for Q Learning and SARSA to properly take place a exploration function must be implemented. The exploration function used was to compare to amount of times each possible location to move to had been visited. The exploration function would then choose the move that would result in the least traveled location unless the location had been visited more than 10 times. 10 is not an important number in this function and was chosen somewhat arbitrarily. In the case that the least traveled location has been visited more than 10 times than we utilize a max reward function instead.

In order to have a fair analysis of both algorithms, the testing grids remained the same from test to test and the amount of trials and episodes remained constant. The amount of trials per test was 500, and 100 episodes per trial. Another thing to mention is for all the grids the move cost was set to -0.2.

The main idea behind these two algorithms is the idea of reinforcement learning by using rewards. Using active agents, we can implement agents that decide on the action based on the given exploration function.

### 3.1.1 Q-Learning

Q Learning is avoids the need for a model by learning based on an action-utility function. By using the Q equation with a Q table we reduce the need for look ahead and allow for the algorithm to simply act by choosing the argmax. The benefit of using a Q Learning algorithm is the lack of a transition model. Q Learning is an off-policy algorithm.

**Tabular Representation** For the tabular representation the agent had implemented a table in order to be able to pull from Q values of areas we had already visited. The table would also be updated as we traverse and determine the new Q values for areas we've already visited as well as areas we haven't. The main benefit of being able to instantly pull values from areas already traversed and the reward state of any other moves, which makes performing comparisons and determining the argmax an easier process.

**Function Approximation** For the function approximation we utilized the function provided on page 804 (22.9, and 22.10). By using the linear function approximator in conjunction with the delta rule, we can determine three different update rules. The benefit is that during the calculation we can reduce the error rate by alpha.

### 3.1.2 SARSA

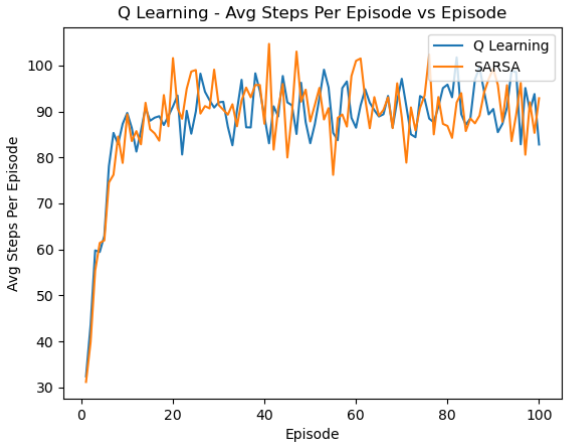
SARSA is very similar to Q Learning except the Q value is determined based on the action taken. SARSA is an on-policy algorithm.

**Tabular Representation** For the tabular representation the agent had implemented a table in order to be able to pull from Q values of areas we had already visited. The table would also be updated as we traverse and determine the new Q values for areas we've already visited as well as areas we haven't. The main benefit of being able to instantly pull values from areas already traversed and the reward state of any other moves, which makes performing comparisons and determining the argmax an easier process.

**Function Approximation** For the function approximation we utilized the function provided on page 804 (22.9, and 22.10). By using the linear function approximator in conjunction with the delta rule, we can determine three different update rules. The benefit is that during the calculation we can reduce the error rate by alpha.

## 3.2 Domains

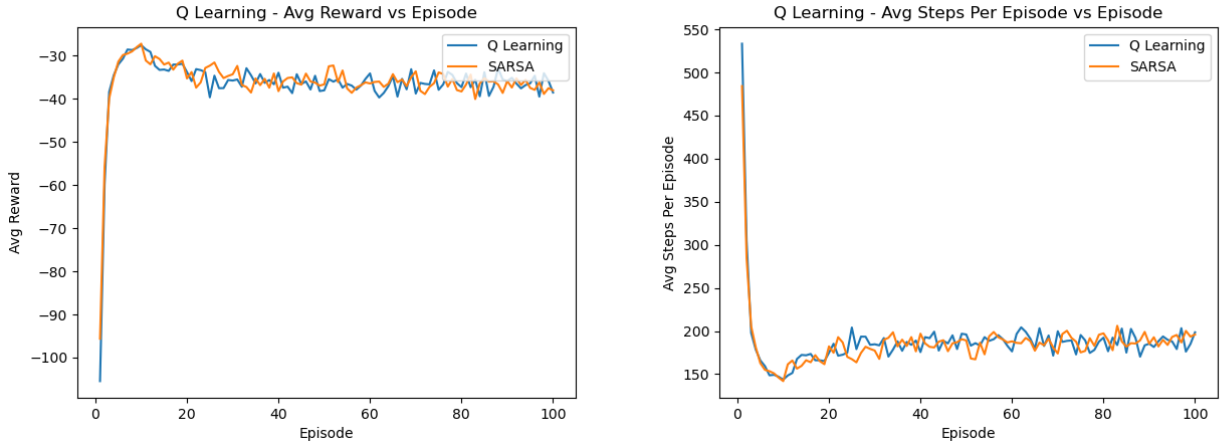
### 3.2.1 4x3 world from Chapter 17

For the comparison between the Tabular and Function approach, The function approach has a a noticeable improvement over the tabular approach but not anything extreme. This is likely due to the small and limiting testing domain of [4,3].

Based on the observed results the comparisons between SARSA and Q Learning are a bit more vague. Q Learning and SARSA had similar runtimes with Q Learning operating slightly faster. When it comes to data comparison SARSA and Q Learning would alternate on which would prioritize the reward. SARSA had the highest and lowest rewards recorded but was not too uniform until the end. With Q Learning the reward was mostly uniform and ended with a higher reward than SARSA. The trade-off is that when one of the algorithms performed better for the reward the same algorithm would also take less steps. This relation makes sense due to the move cost and additional moves reducing the reward.

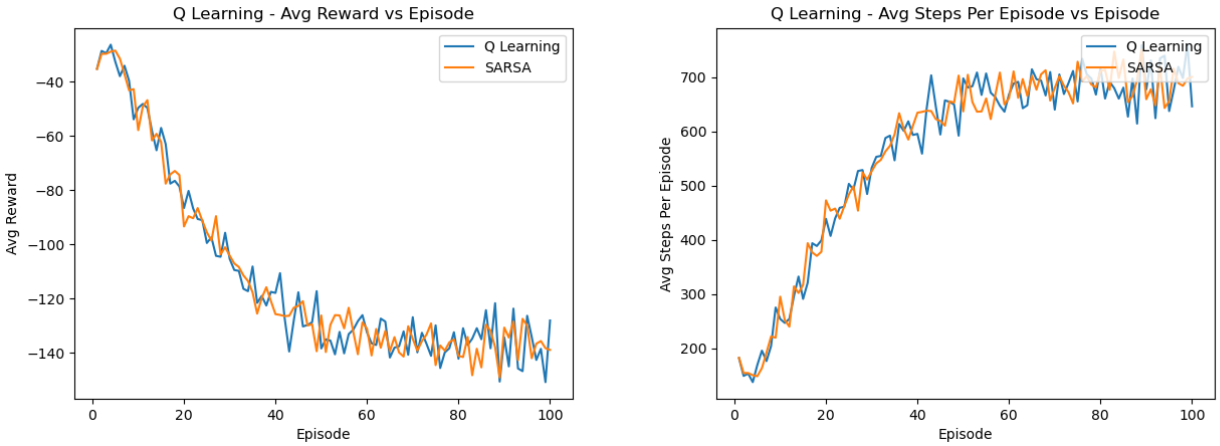
#### 3.2.2 10x10 with no obstacles, and a +1 reward at (10,10)




For the comparison between the Tabular and Function approach, the function approach excels in this scenario since the path is essentially a diagonal line. This makes the function approach nearly linear and very efficient.

The recorded data is very uniform in this domain. This is due to the linear relationship between the starting point and the goal position. There is a similar relationship between SARSA and Q Learning as described in the previous domain. based on the data recorded it can be concluded that both algorithms are acting optimally after 20 episodes.

### 3.2.3 10x10 with no obstacles, and a +1 reward at (5,5)




For the comparison between the Tabular and Function approach, the function approach could be seen as an absolute failure. this is mostly due to the reward at [5, 5] which makes the utility similar to a pyramid. In order to solve this we can add a feature to determine the distance to goal which improves the performance of the linear approximator.

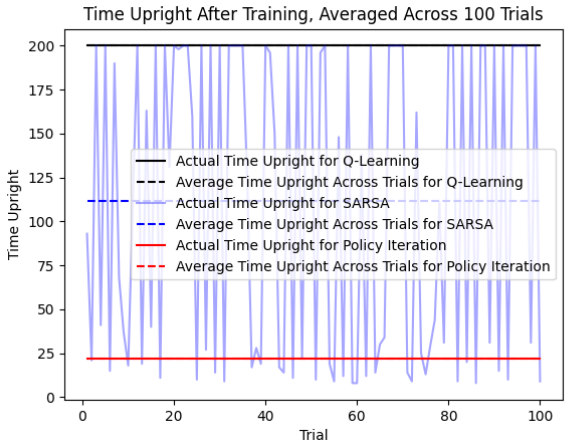
In the last domain unlike how the previous domain was optimal this domain is optimal. The reasons why are explained from the tabular and function comparison. By looking at all the observed data it can be seen that in 500 trials we fail to reach an optimal uniform traversal. One of the main differences is while in the 4x3 example the SARSA algorithm varied more than Q Learning algorithm, in this case the Q Learning algorithm has a larger standard deviation than the SARSA algorithm.

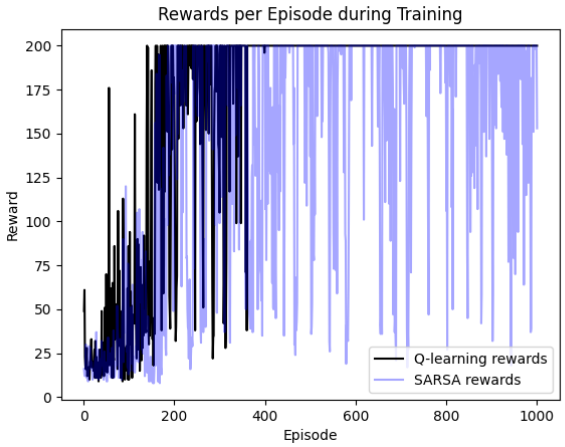
### 3.2.4 Cartpole

Similar to the Policy and Value Iteration section, we utilized the OpenAI gym cart-pole environment for the cartpole domain. Regarding implementation, we utilized the Q-Learning solution and the SARSA solution that interfaces with the cart-pole environment provided by OpenAI. This code can be located at https://github.com/init-22/CartPole-v0-using-Q-learning-SARSA-and-DNN , and it is included as a git submodule in our submitted project code. To allow for comparable results between both the Q-Learning and SARSA solutions and with the Policy Iteration solution using the same OpenAI gym cart-pole

environment, we kept the parameters consistent. We utilized 1000 episodes, and measured the rewards per episode, as well as the time upright after training was complete. We also have decay set to 25, learning rate set to 0.1, and epsilon also set to 0.1. There is a slight difference between to the two solutions, in that the Q-Learning uses a discount of 1.0, and SARSA uses 0.98. This difference will be discussed further into this report. In regards to episode length, we set our goal to remain upright for 200 time steps. Rewards were equivalent to the number of time steps the pole stayed upright. For instance, if the pole remained upright for 25 time steps, the reward would be 25.

For analysis of this specific domain for this implementation, images can be seen below.




From the reward vs episode image, it is evident that the SARSA algorithm was able to reach the maximum reward after 400 episodes, whereas the Q-Learning algorithm jumped around substantially more. This is due to a few factors. For one, the environment we utilized for this portion of the project was not ideal. One main contributing factor was that the Q-Learning implementation used a discount factor of 1.0, which led to poor decision-making and poor state evaluation. Once the agent was trained it could stay upright for the full 200 time steps for SARSA, however observing the rendering of the Q-Learning cartpole vs the SARSA cartpole, you can see that the Q-Learning cartpole appears more "sloppy", and Q-learning was not able to stay upright for the full 200 time steps. The SARSA cartpole rendering appears as if it can easily outlast the 200 timestep period.

The second factor is in regards to the algorithmic differences in Q-Learning and SARSA. Since SARSA is an on-policy algorithm and bases its evaluation on actions taken, it can get a better grasp of optimal actions compared to the

greedy-like approach of Q-Learning which estimates rewards for future actions. This does not mean that SARSA always outperforms Q-Learning, but for this specific implementation with this specific domain, it does.

Lastly, the third factor is in the reward scheme. In this implementation, rewards were biased to favor SARSA more-so than Q-Learning. Since SARSA can definitively know its rewards based on how long the agent in training was staying upright, optimal actions could be determined more readily. Since Q-Learning is off-policy and relies on reward estimation, it did not have the benefit of being able to definitely know exactly how long the agent in training would be upright for. In this instance where the pole may be upright for 200 time steps, that can fall a bit outside the ability of Q-Learning to make accurate estimate for a time so far into the future.

However, despite the roughness in training, both Q-Learning and SARSA were able to achieve their goals of staying upright for 200 time steps. Since training was conducted with 1000 episodes, there was ample time for both agents to be adequately changed. If we were to repeat this project, altering the number of episodes like we did for the base domain might yield more interesting results, since we could see how the number of episodes affected the quality of the training for each implementation.

As a note regarding the resulting image, the Policy Iteration upright time appears to be flat at a value of 20. The resulting data is correct and is upright for the full 200 timesteps, but the environment is off by a factor of 10 when plotting the results, making the resulting image appear as 20 instead of 200. The environment has since been adjusted, but due to the lengthy runtime of the program, new data was not collected.

# 4 Multi-Armed Bandit

## 4.1 Domain and Data Description

For this portion of the project, the base domain was to evaluate the epsilon-greedy algorithm and a UCB algorithm in terms of regret. Due to the larger group size, our additional domain included altering the standard deviation, and providing brief analysis on convergence. Due to the vast amount of figures, they have been stored in the Appendix. The implementation of the UCB algorithm was quite simple, which was expected per the lecture discussion. This was especially true when coupled with the provided Bandit Simulator from the TA, Robert Geraghty, where the arm of the bandit could be pulled with only a few lines of code. As a result of the simple implementation, extra work was

performed in terms of data collection to provide better analysis.

For collecting data, a few parameters were altered. The number of samples, the number of arms, and epsilon were all varied for the base domain. For each parameter instance, the UCB and epsilon-greedy algorithms were ran 100 times, and results were averaged. As a result, especially when viewing data on the epsilon-greedy algorithm, there will not be many large spikes as you might expect with an algorithm running fairly randomly. Regardless of not seeing the large spikes, the randomness still pushes the data trend downwards during the averaging process, and it is verified that the worst arms are still pulled when viewing the figures displaying the number of pulls on each arm.

Issues with some of the data collection: For the purpose of the report, we do have a very large number of figures. To try and show vastly different number of results, we limited ourselves to fewer parameters, such as 3, 5, 10, 50, and 99 arms, and 100 and 1000 samples. When looking at our figures, it does appear that our UCB algorithm can perform quite poorly. This is largely due to our poor selection in parameters. 99 arms and both 100 and 1000 samples is simply a losing scenario for the UCB algorithm. With the large number of arms and not enough samples, the epsilon-greedy algorithm will of course outperform it in almost every case. While this specific data is helpful to demonstrate the importance of the number of samples vs the number of actions, if we were to repeat this project, we would likely have more "winning" cases for the UCB by ensuring that it had enough time to show convergence and its results from said convergence.

## 4.2 Analysis

In terms of analysis, the UCB algorithm outperforms the epsilon-greedy algorithm in most cases, though there are a few conditions to this statement. Primarily, the multiple (m) of samples and actions must be large enough. Since the UCB algorithm attempts to balance exploration and exploitation with enough exploration early on, if the algorithm pulls most of the arms early and has little samples left to exploit the better arms, it cannot adequately converge. This is seen in the figures where there are 99 arms on the bandit, and in cases where there are only 100 samples taken. Since all the arms are pulled, the UCB algorithm does not have enough time to converge. Even in the case with 1000 samples and 99 arms, the multiple is not enough for convergence. The convergence line does drift upward over time, however there are simply not enough samples to overcome the excessive exploration that is done. This is seen through the figures displaying the number of arms pulled for the UCB algorithm.

In this specific implementation, there are a bit of biases that aid the case of epsilon-greedy as well. When there is a tie in the payout average, the epsilon-

greedy algorithm just chooses the last arm to pull. In this case, the last arm is the one with the best payoff. This results in the epsilon-greedy algorithm deciding early on that the last arms are the best to pull, and can converge very rapidly, whereas the UCB algorithm must explore more. If arm payouts were randomized, the differences in UCB and epsilon-greedy would be much more apparent. This can be noticed when epsilon is equal to 0.3. Since there is now a 30% chance that a random arm will be pulled, the epsilon-greedy algorithm is not able to exploit on its early convergence, pushing the overall payout downward. Once the UCB algorithm converges, the 30% randomness from the epsilon-greedy algorithm cannot compete in terms of regret. When looking especially at the 1000 sample cases when arms are less than 50, the UCB algorithm performs substantially better since the payouts are much more reliable.

### 4.3 Additional Domains

When altering the standard deviation, there are some interesting results. The minimum reward is bounded as follows:

$$\mu - \sigma\sqrt{h-1} \leq minimum\_payout \leq \mu - \frac{\sigma}{\sqrt{h-1}}$$

Where $\mu$ is the mean, $\sigma$ is the standard deviation, and h is the number of samples. The maximum value is bounded on its maximum as follows:

$$maximum\_payout \leq \mu + \sigma\sqrt{h-1}$$

When decreasing the standard deviation, the payouts trend closer towards their means. Likewise, when increasing the standard deviation, the payouts drift farther from their means. In the Bandit Simulation provided, the arm payouts are determined based on the number of arms that exist. Theoretically, for systems with fewer arms, there is not a substantial difference other than that convergence trends closer towards the actual reward means when the standard deviation is lessened. From the data collected, it is seen that this is true for both epsilon-greedy and for the UCB algorithm.

When increasing the standard deviation however, the epsilon-greedy algorithm begins to suffer. The UCB algorithm, once converged, starts to get much greater payout. Since the algorithm has converged, there is excess capitalization since pulling the best arm repeatedly yields results where the maximum payout can receive much higher rewards than the mean suggests. However with the epsilon-greedy algorithm, it cannot determine the best arm to pull. This becomes increasingly true when the number of arms increase. Since there is now overlap in possible reward (since the arm means + standard deviation can now cross territories with other arm means), the epsilon-greedy algorithm's greedy

approach starts to falter. For instance, take a parameter set of standard deviation = 0.2, samples = 100. Let's compare two arms, one with mean of 0.5, and one with mean 0.6. At its extremes, the arm with means 0.5 can yield a reward of 2.49, and the arm with means 0.6 can receive a reward of -1.99. While these values are the extremes, it shows that the epsilon-greedy algorithm now will believe that the arm with lower means is the better option, and can get stuck pulling on that arm instead of trying to pull the arm with a higher means.

# 5 Conclusions

## 5.1 Value Iteration

The main advantage to utilizing Value Iteration is in its simplicity. Like the Policy Iteration implementation, it starts with a random value function and is guaranteed to converge. Implementing the algorithm is beneficial due to its simplicity, but it does suffer as a result. Since the algorithm runs through all possible actions in the state space and must identify the maximum reward, it is computationally expensive when there are many actions to check. This also leads the algorithm to a slower runtime, when this step must be repeated until convergence. In addition, the Value Iteration algorithm requires additional steps to converge. However, value iteration is still beneficial since it is still guaranteed to converge, and has a straight-forward implementation.

## 5.2 Policy Iteration

Policy Iteration is advantageous is that it can identify an optimal policy. The downside of the Policy Iteration algorithm is that it suffers from poor performance in terms of runtime, especially when the size of the problem and complexity of the problem increases. In our testing Policy iteration ran worse on wumpus world due to the increased complexity, and as we increased the size of the problem. Despite its difficulties, in scenarios where there are a finite number of states, and the number of states is relatively small, then Policy Iteration is a viable implementation for determining a policy. As the number of states and complexity increases, switching to another form of learning could yield better results.

## 5.3 Q-Learning

The Q Learning algorithm is useful as without the need for a model based implementation and the ability to operate without the need of a transition model. In Q Learning we apply a action-utility function in order to determine the discounted reward if the agent takes an action. By being able to determine the discounted reward we can simply choose an action by using the argmax. However, if we only use the argmax it is likely we will get stuck in a loop and fail to explore the map. In order to circumvent this, we use a exploration function to ensure traversal to unknown areas.

As the Q Learning algorithm is performed we choose the area least visited unless all the areas have been visited enough, in that case we then chose where to traverse based on the argmax. Based on our experimentation while Q Learning isn't the most efficient or optimal algorithm for the job it is still able to accomplish what is required in a somewhat efficient manor. The differences between running the algorithm with a tabular and function approximator agent are noticeable and can be detrimental in some cases but since the function approximation can have additional features to determine the distance it is less of an issue.

## 5.4 SARSA

The SARSA algorithm is very similar to the Q Learning algorithm with the main difference being that we calculate the Q value of the action we decide to perform instead of comparing the Q value of the actions we could take. In our testing we determined that the SARSA algorithm ran faster and overall showed better and more consistent results than the Q Learning implementation. Overall for the gridworld implementation the Q Learning and SARSA were pretty equal in terms of episodes vs rewards, however during the cartpole domain the SARSA algorithm vastly outperformed the Q Learning implementation.

## 5.5 UCB

The UCB algorithm is advantageous to use in that it is a relatively simple algorithm to implement, yet can provide more optimal results than a typical greedy or even the $\epsilon$-greedy approach. Like discussed in the respective analysis portion however, there are a few points to consider. The greedy and $\epsilon$-greedy strategies can outperform the UCB algorithm if the multiple, $m$, of the number of actions and the number of samples is not of adequate size. While the UCB must balance exploration and exploitation, the $\epsilon$-greedy algorithm does not focus

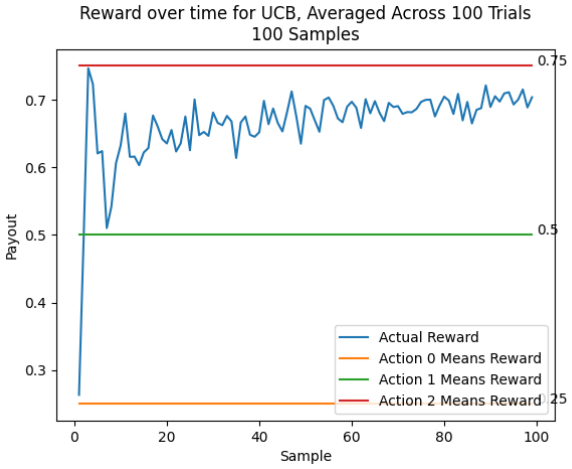
on exploration. As a result, if there are too few samples to properly explore all arms, then the UCB cannot converge, whereas the $\epsilon$-greedy algorithm can start to exploit some of the higher payout actions. The $\epsilon$ value also plays a crucial role. Since the $\epsilon$-greedy algorithm can get stuck, the randomness can help force the algorithm to exploit different actions. Compared to UCB however, if the $\epsilon$ value was too high and UCB was able to quickly converge, then the UCB algorithm could exploit a higher payout action and outperform the $\epsilon$-greedy algorithm which was forced to explore different actions with up to a 30% chance.

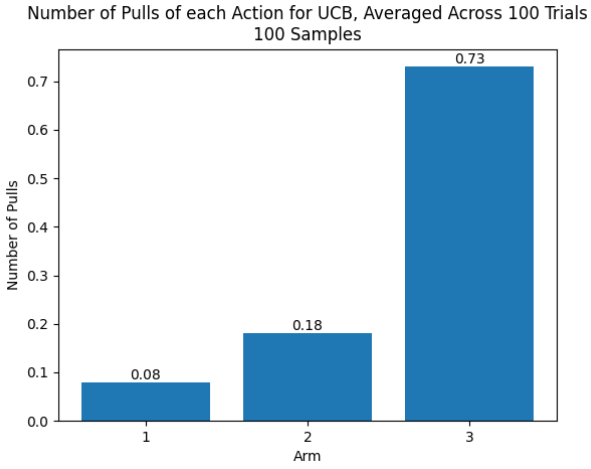
Lastly, the standard deviation also plays a role. As discussed during the analysis section, when there is a high number of arms and the standard deviation is large, there are overlaps in rewards of actions. As a result, the $\epsilon$-greedy algorithm can get stuck on a lower payout action. Assuming the UCB algorithm has enough time to converge, a higher standard deviation can lead to much greater reward since it can exploit the best action and receive greater maximum payouts. In systems with large number of actions, having tighter bounds (smaller standard deviation) can aid both algorithms to reach convergence faster. Since there will be less overlap in payouts and rewards will be closer to the true means reward of actions, the algorithms can identify the best action easier.

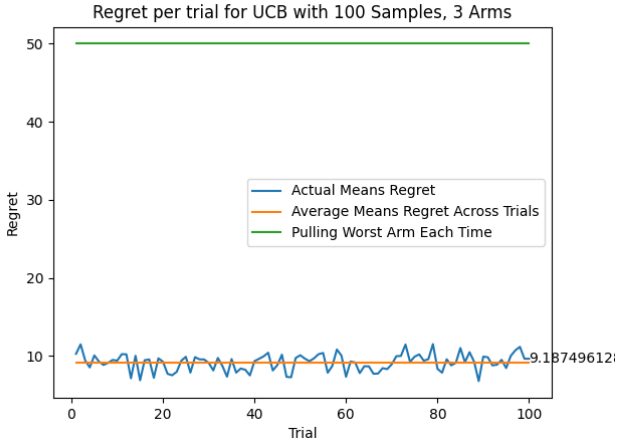
# 6 Appendix

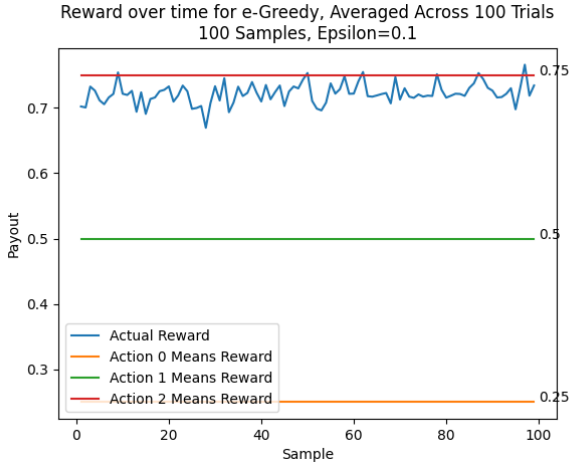
## 6.1 Base Domain: Standard Deviation = 0.1

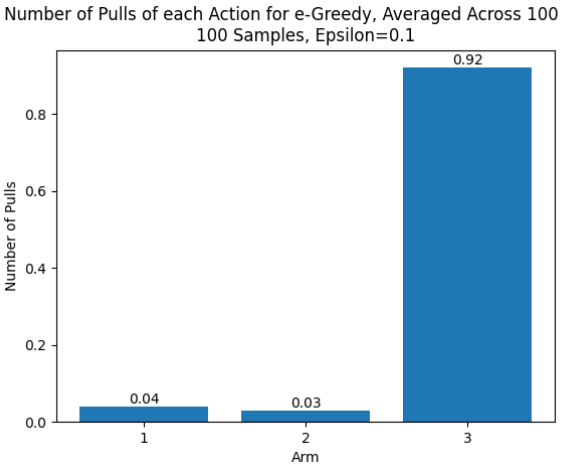
### 6.1.1 100 Samples, 100 Trials, 3 Arms, Std Dev=0.1

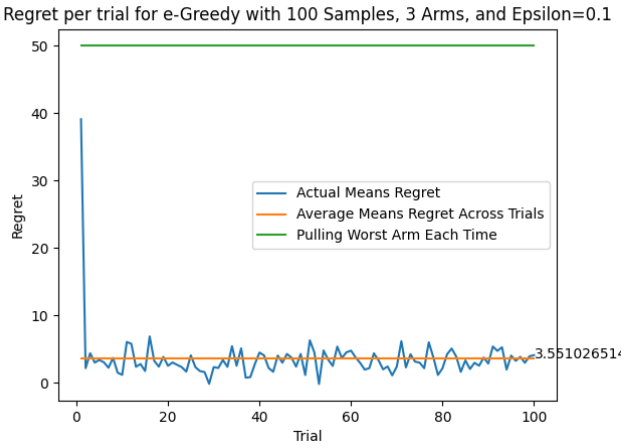


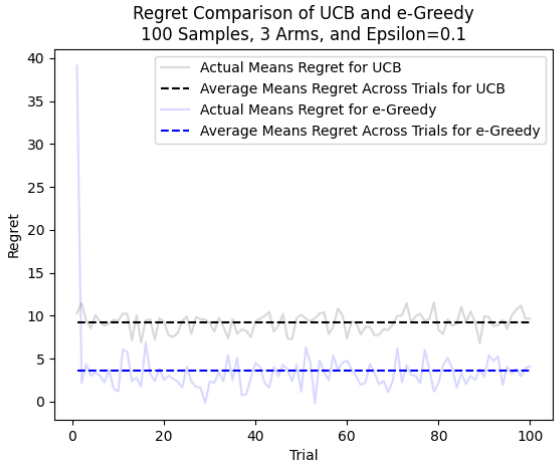



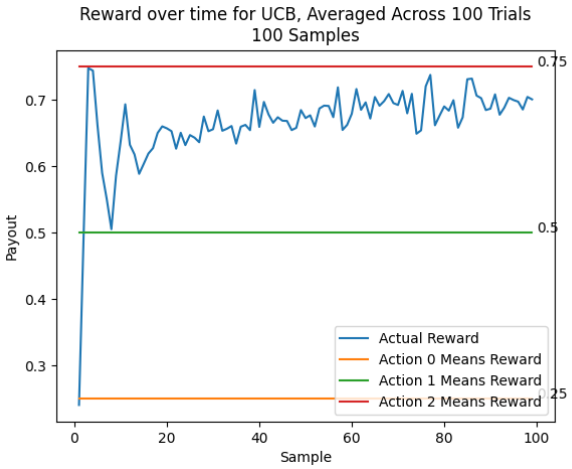



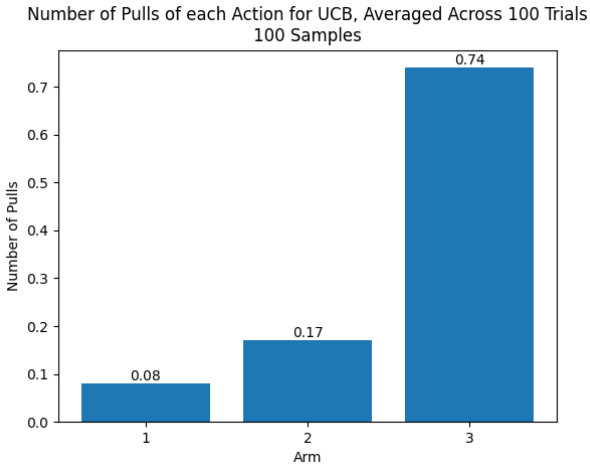


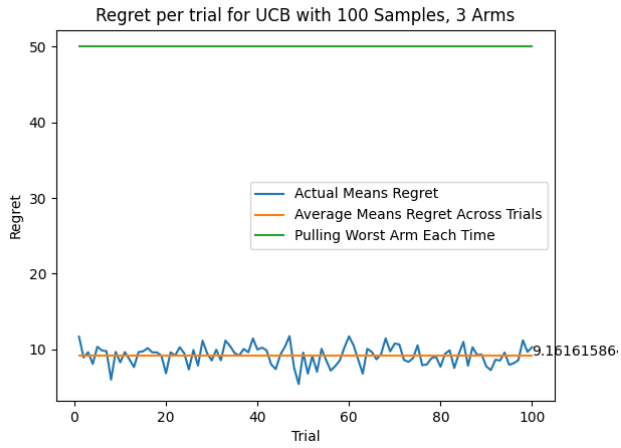



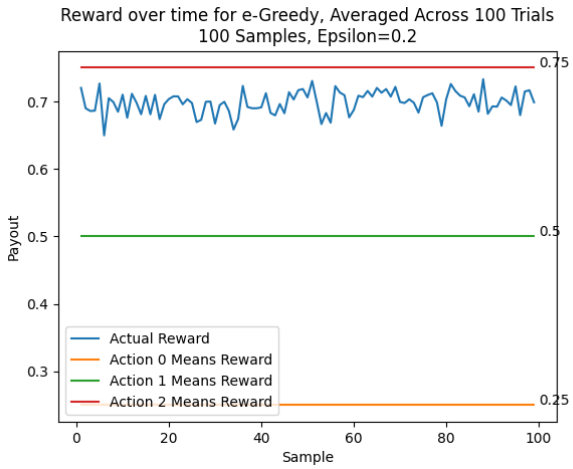



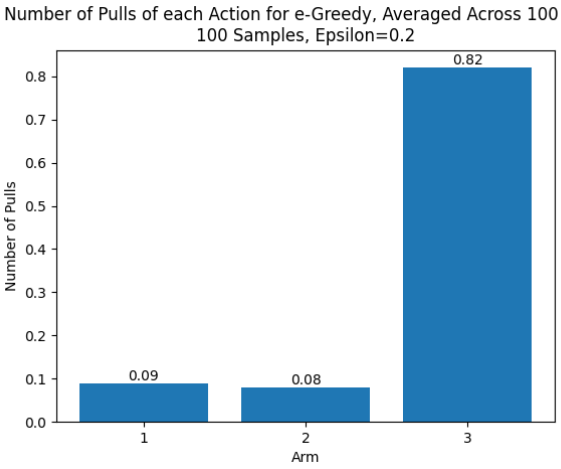

**100 Samples, 100 Trials, 3 Arms, Epsilon=0.1 Std Dev=0.1**

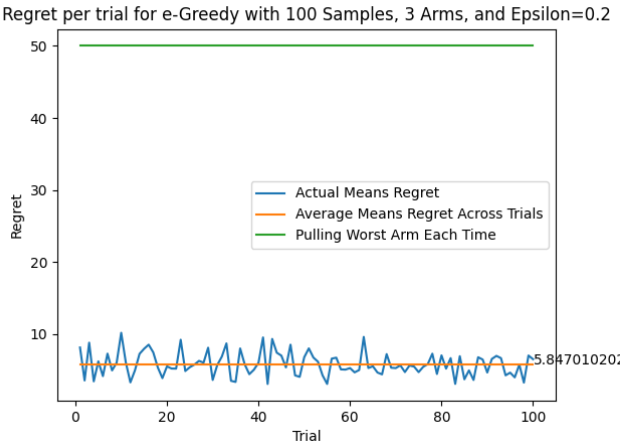

Reward over time for UCB, Averaged Across 100 Trials
100 Samples
Payout
Sample
Actual Reward
Action 0 Means Reward
Action 1 Means Reward
Action 2 Means Reward
0.75
0.5
0.25

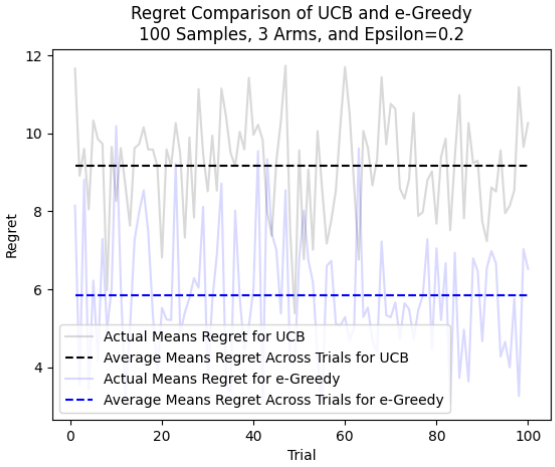

Number of Pulls of each Action for UCB, Averaged Across 100 Trials
100 Samples
Number of Pulls
Arm
0.08
0.17
0.74


Regret per trial for UCB with 100 Samples, 3 Arms
Regret
Trial
Actual Means Regret
Average Means Regret Across Trials
Pulling Worst Arm Each Time
9.16161586

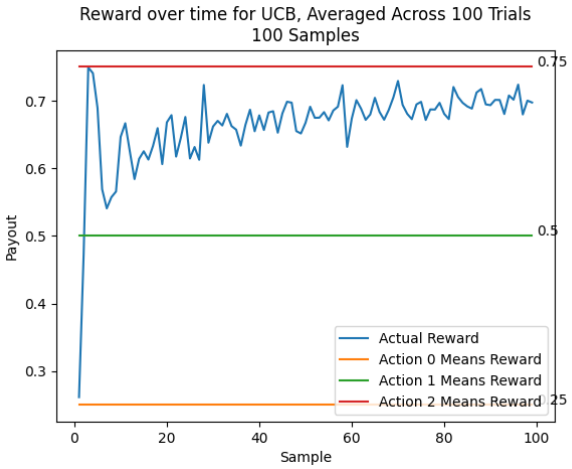

Reward over time for e-Greedy, Averaged Across 100 Trials
100 Samples, Epsilon=0.2
Payout
Sample
Actual Reward
Action 0 Means Reward
Action 1 Means Reward
Action 2 Means Reward
0.75
0.5
0.25

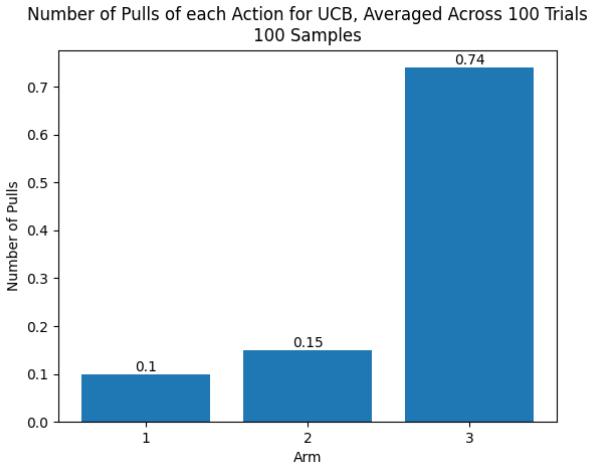


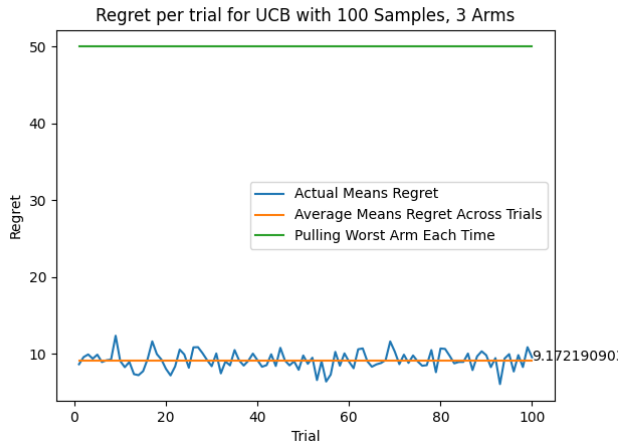



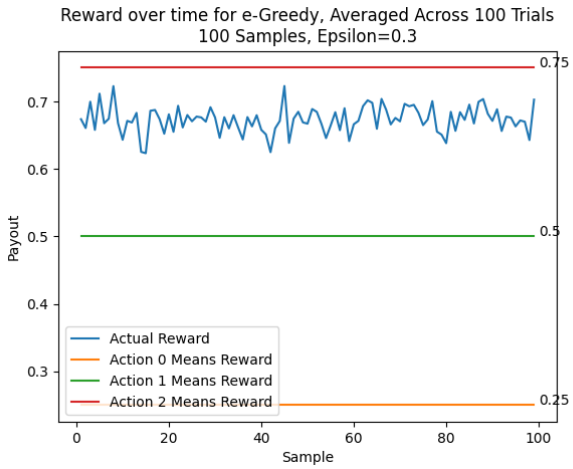



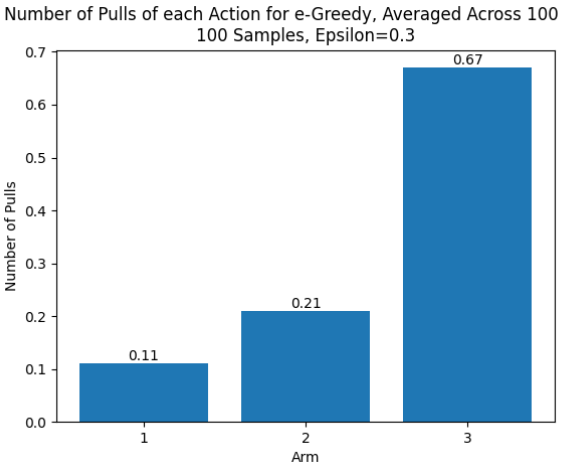

**100 Samples, 100 Trials, 3 Arms, Epsilon=0.2 Std Dev=0.1**

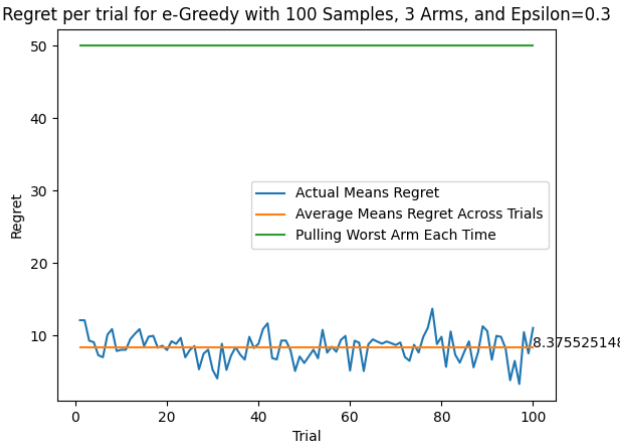

Reward over time for UCB, Averaged Across 100 Trials
100 Samples
Payout
Sample
Actual Reward
Action 0 Means Reward
Action 1 Means Reward
Action 2 Means Reward
0.75
0.5
0.25

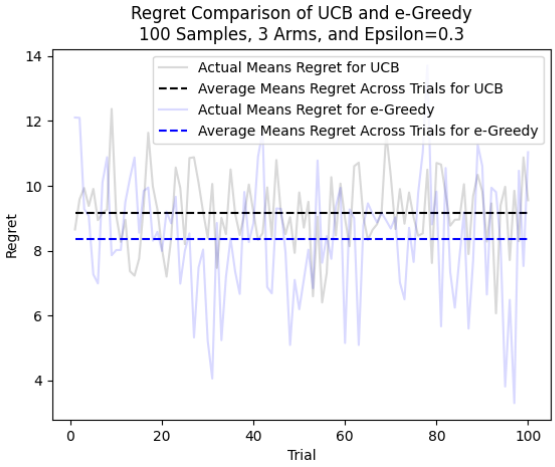

Number of Pulls of each Action for UCB, Averaged Across 100 Trials
100 Samples
Number of Pulls
Arm
0.1
0.15
0.74


Regret per trial for UCB with 100 Samples, 3 Arms
Regret
Trial
Actual Means Regret
Average Means Regret Across Trials
Pulling Worst Arm Each Time


Reward over time for e-Greedy, Averaged Across 100 Trials
100 Samples, Epsilon=0.3
Payout
Sample
Actual Reward
Action 0 Means Reward
Action 1 Means Reward
Action 2 Means Reward
0.75
0.5
0.25

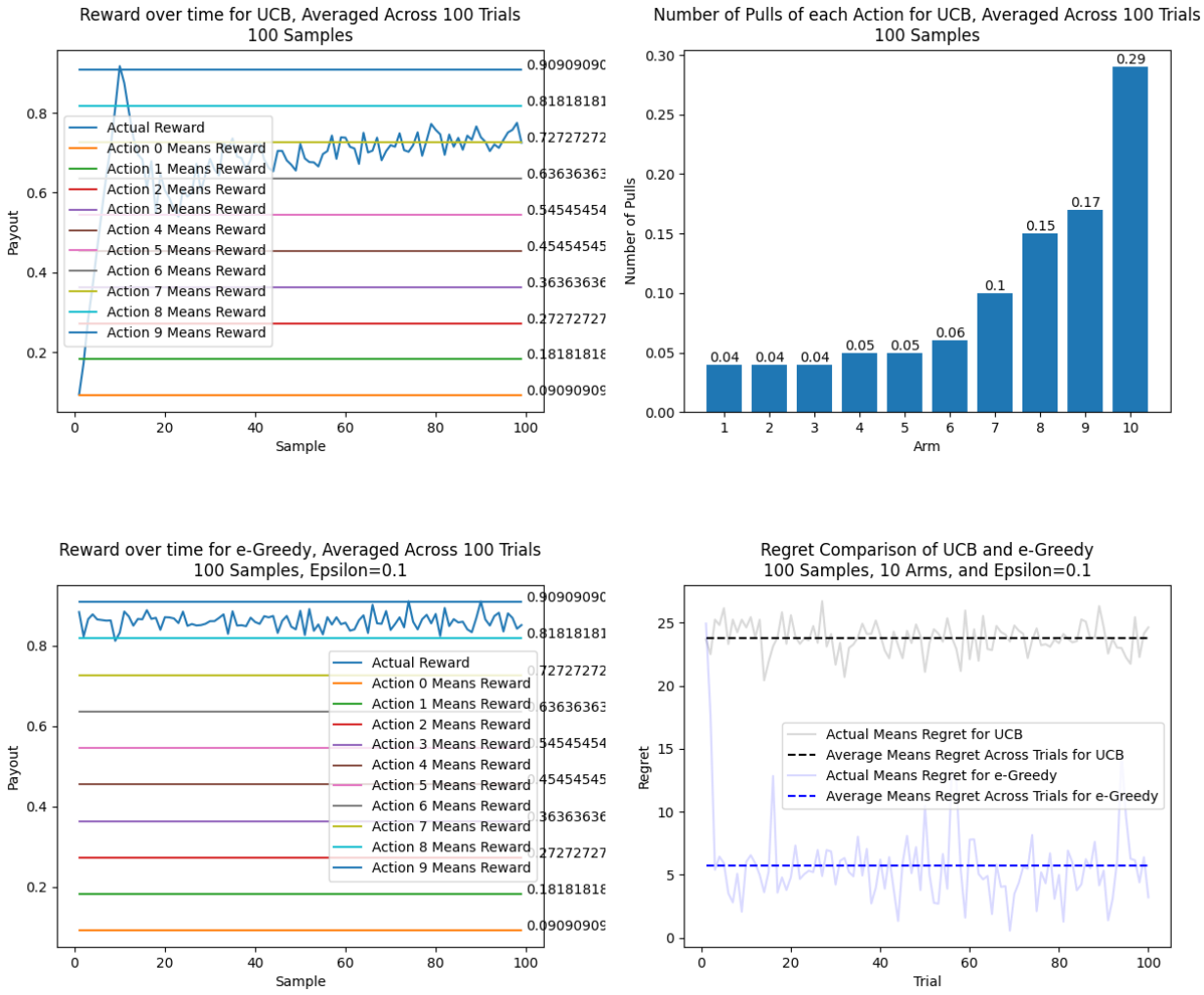


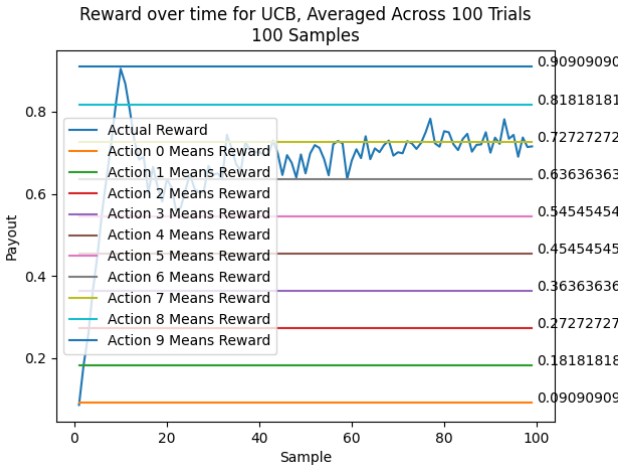



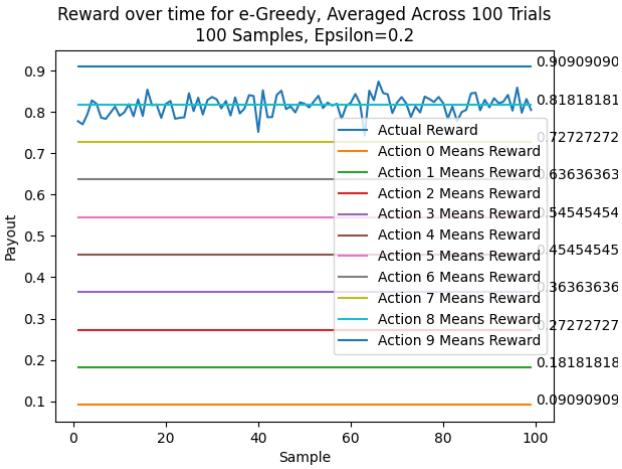

**100 Samples, 100 Trials, 3 Arms, Epsilon=0.3 Std Dev=0.1**

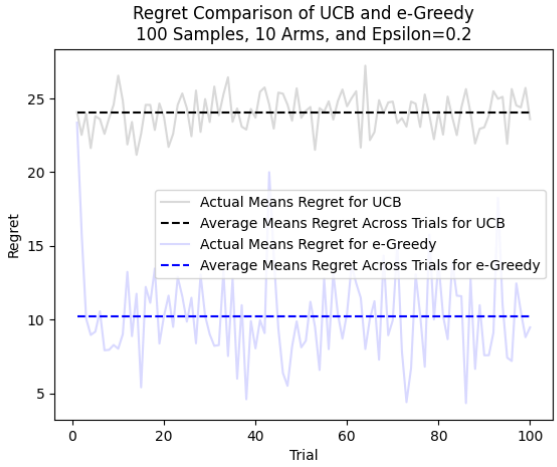
### 6.1.2 100 Samples, 100 Trials, 10 Arms, Std Dev=0.1



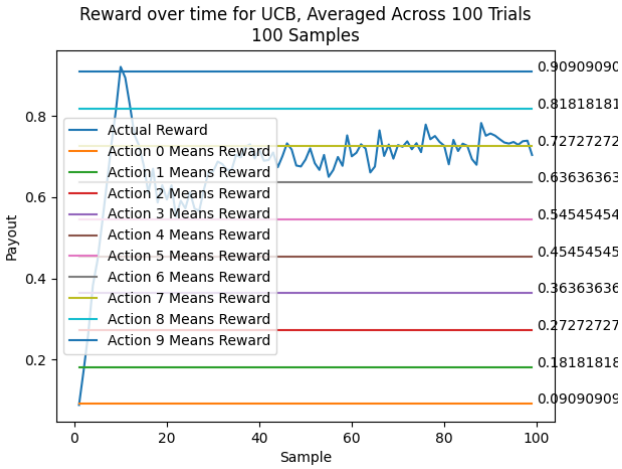

**100 Samples, 100 Trials, 10 Arms, Epsilon=0.1 Std Dev=0.1**

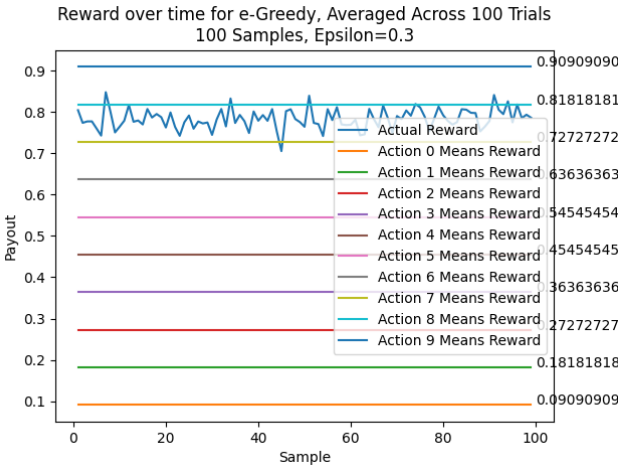


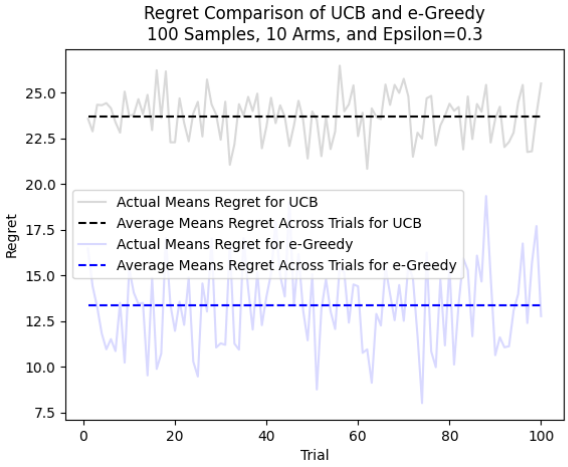




**100 Samples, 100 Trials, 10 Arms, Epsilon=0.2 Std Dev=0.1**

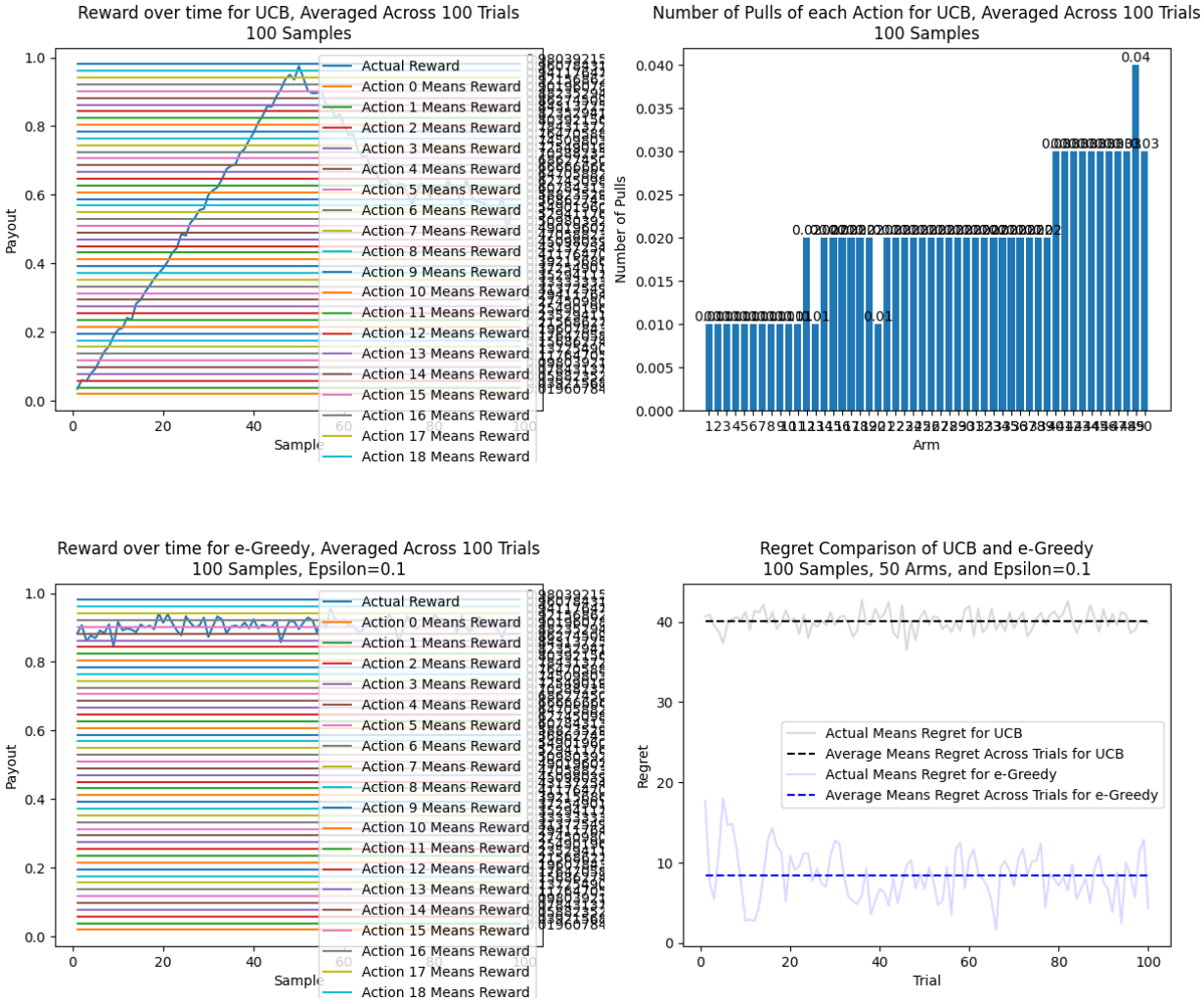


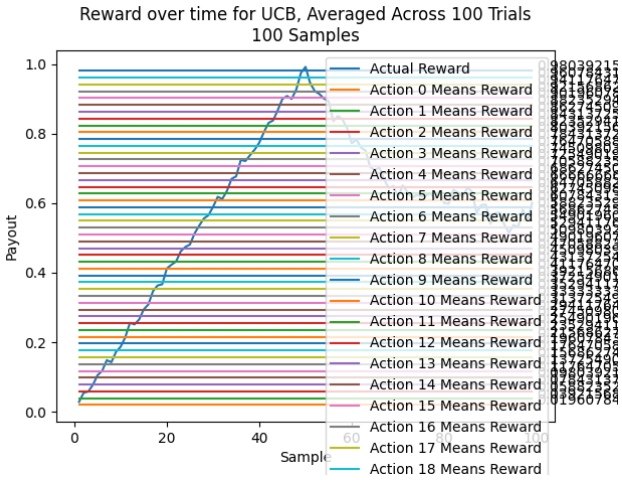



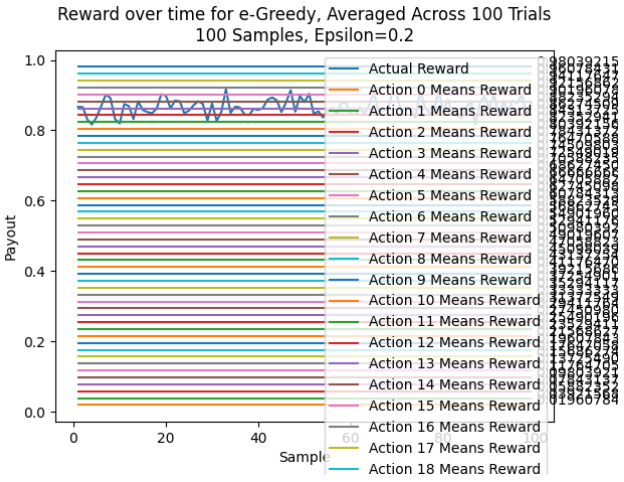

**100 Samples, 100 Trials, 10 Arms, Epsilon=0.3 Std Dev=0.1**

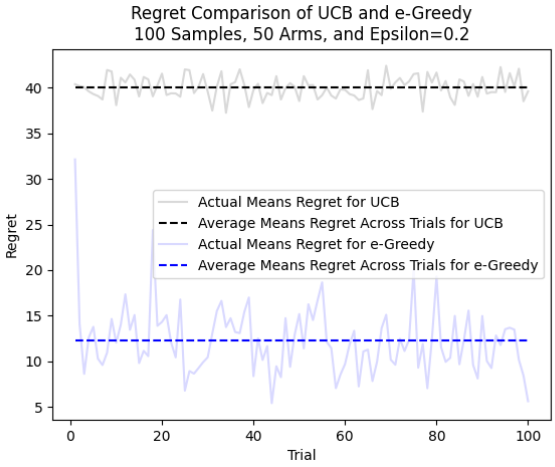
### 6.1.3 100 Samples, 100 Trials, 50 Arms, Std Dev=0.1



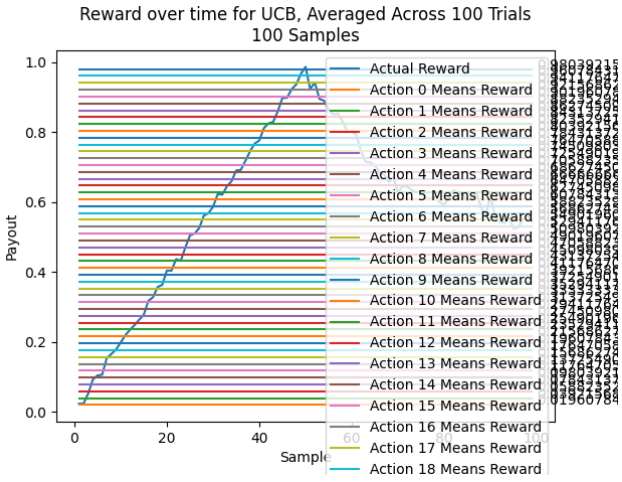

**100 Samples, 100 Trials, 50 Arms, Epsilon=0.1 Std Dev=0.1**

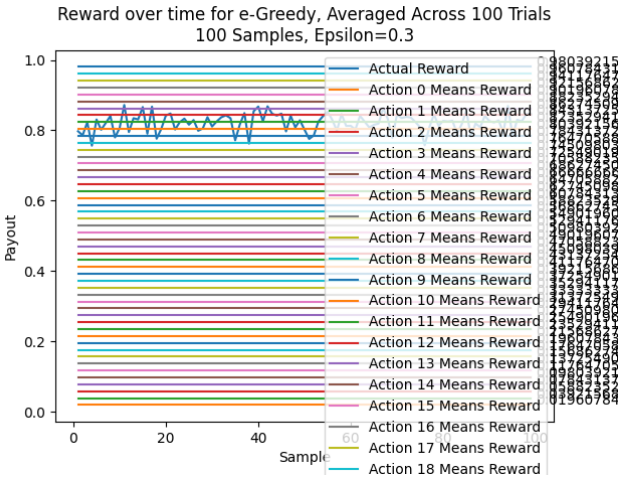


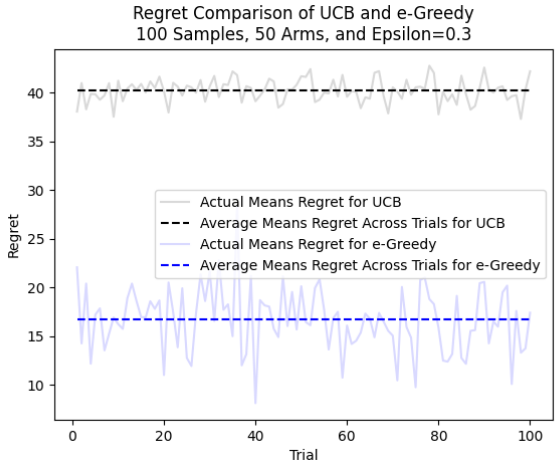




**100 Samples, 100 Trials, 50 Arms, Epsilon=0.2 Std Dev=0.1**

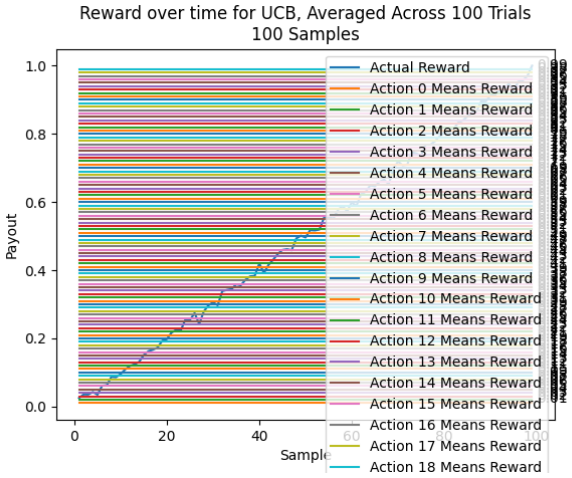


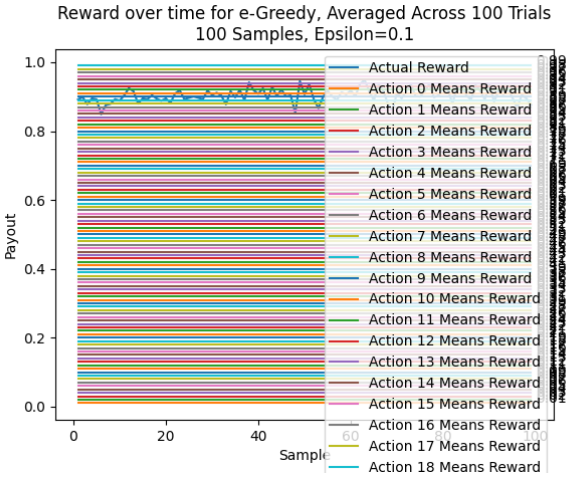



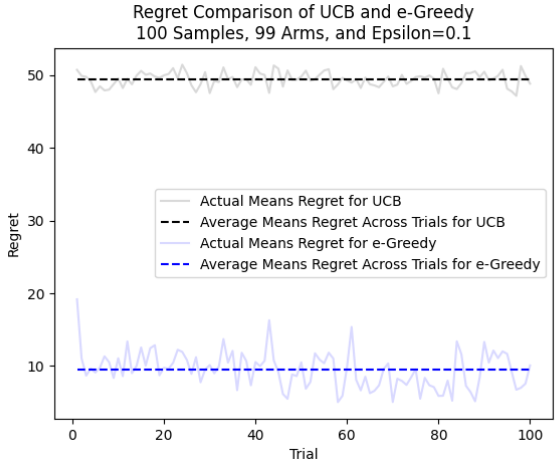




**100 Samples, 100 Trials, 50 Arms, Epsilon=0.3 Std Dev=0.1**

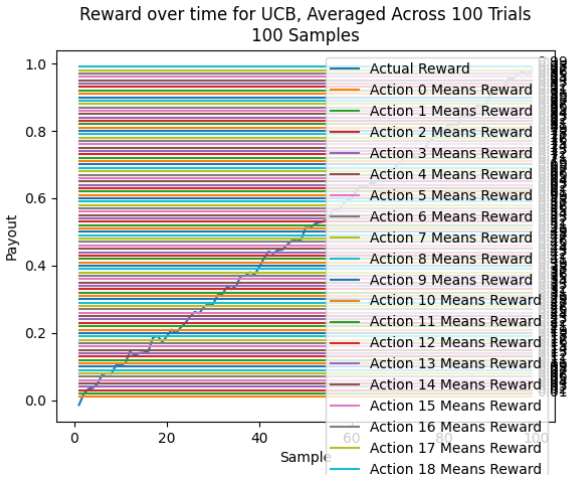
### 6.1.4 100 Samples, 100 Trials, 99 Arms, Std Dev=0.1

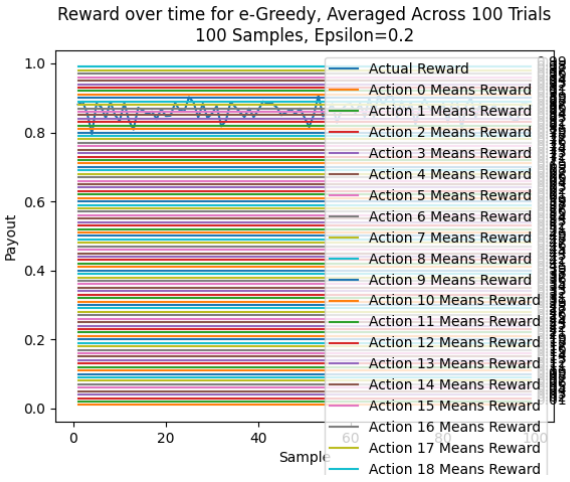


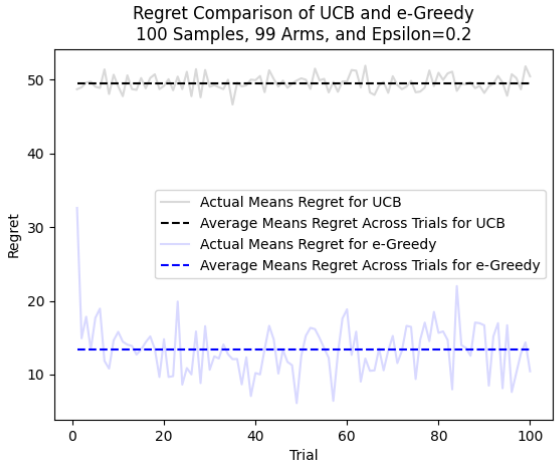



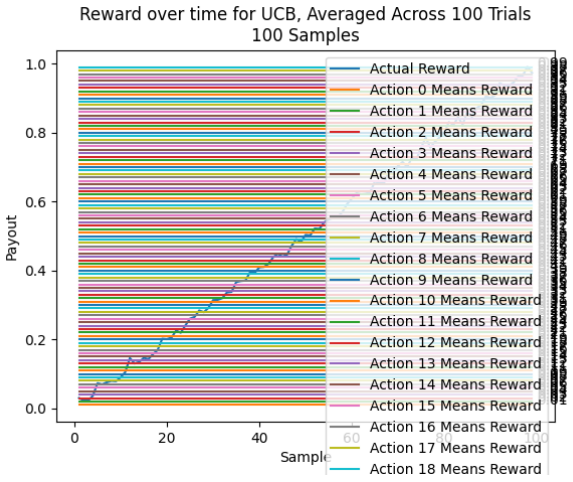

**100 Samples, 100 Trials, 99 Arms, Epsilon=0.1 Std Dev=0.1**

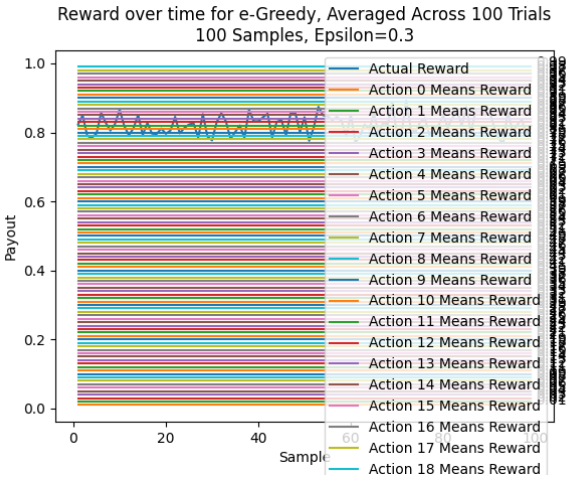


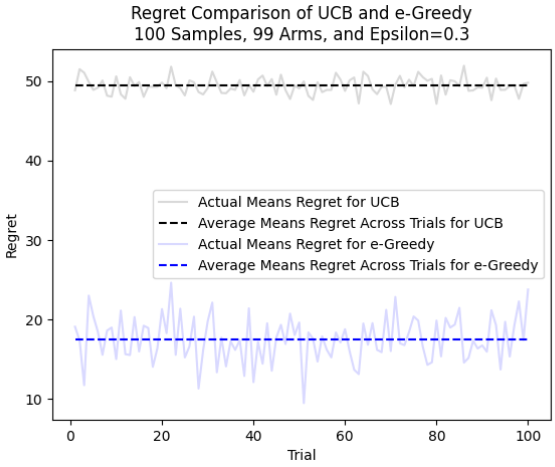




**100 Samples, 100 Trials, 99 Arms, Epsilon=0.2 Std Dev=0.1**

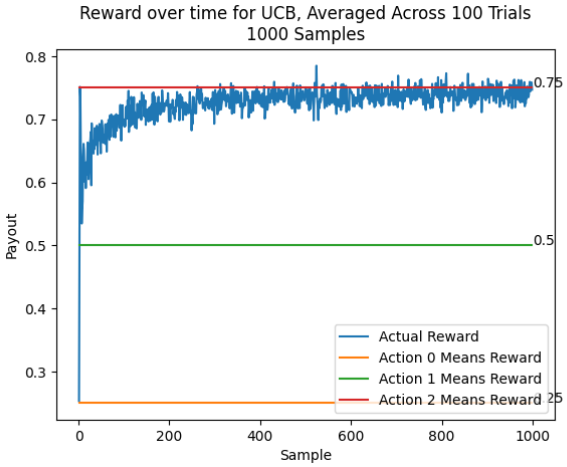


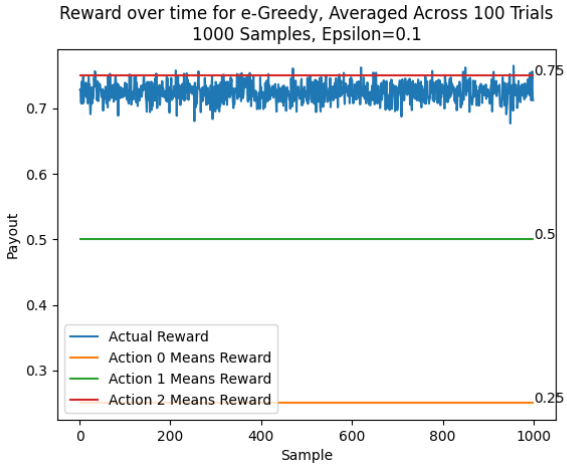



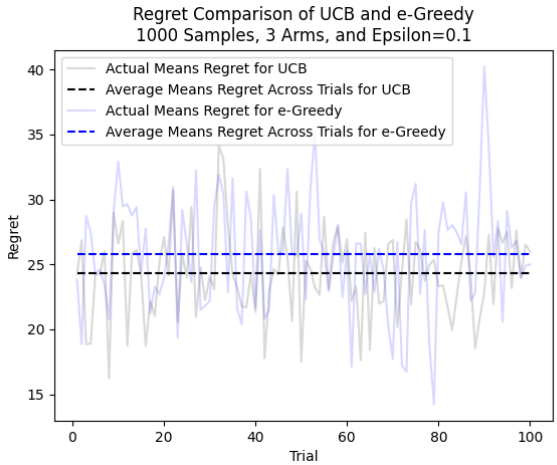




**100 Samples, 100 Trials, 99 Arms, Epsilon=0.3 Std Dev=0.1**

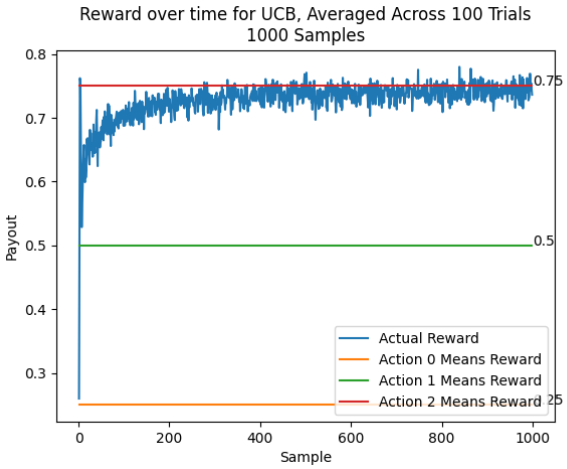
### 6.1.5 1000 Samples, 100 Trials, 3 Arms, Std Dev=0.1

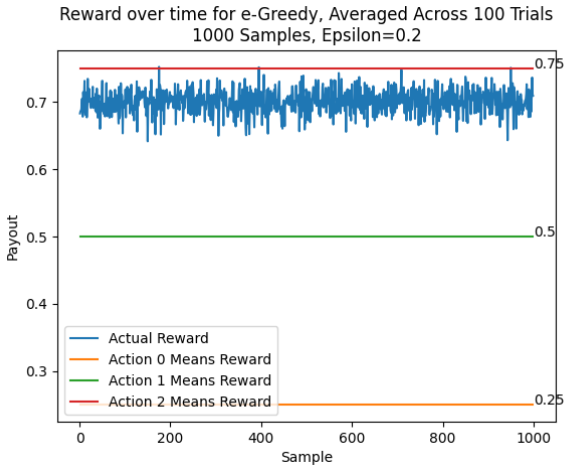


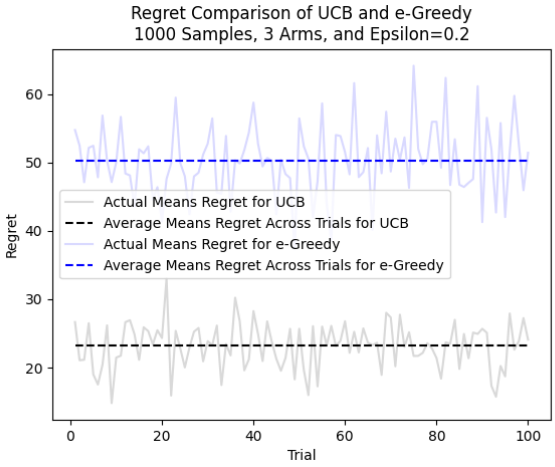



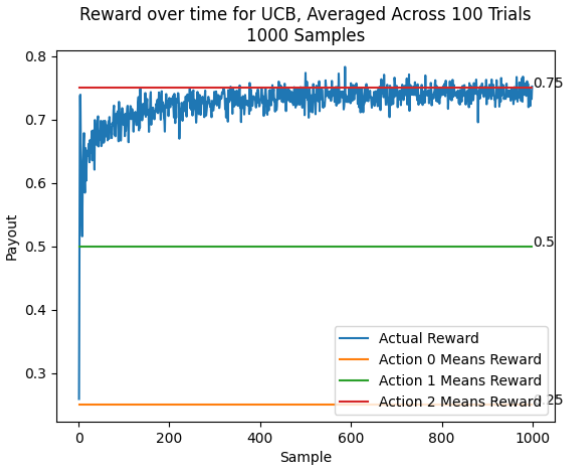

**1000 Samples, 100 Trials, 3 Arms, Epsilon=0.1 Std Dev=0.1**

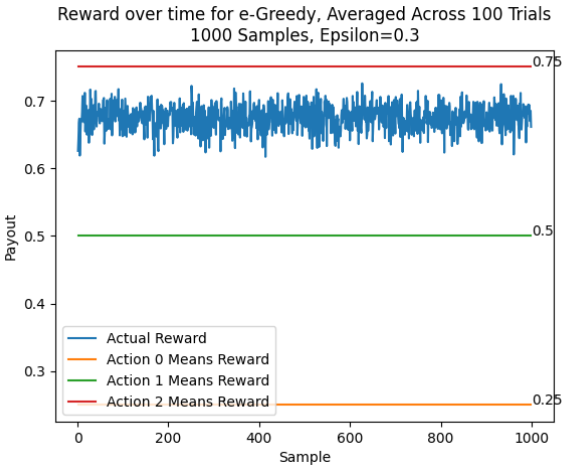


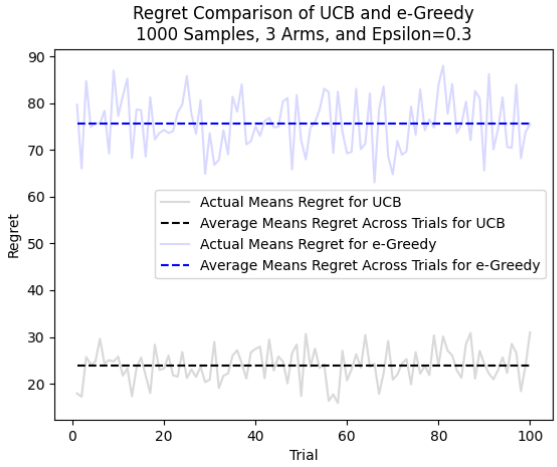




**1000 Samples, 100 Trials, 3 Arms, Epsilon=0.2 Std Dev=0.1**

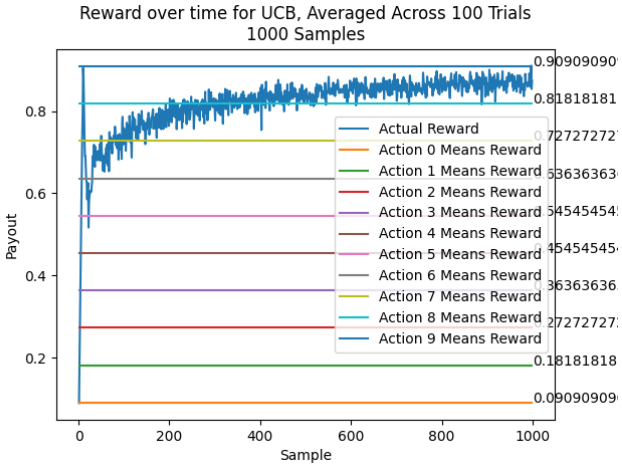


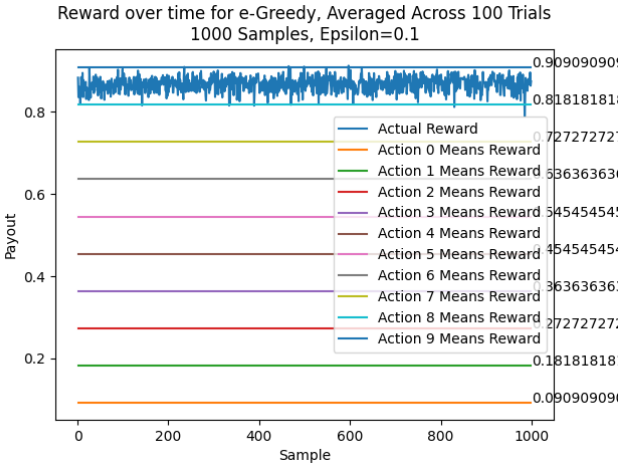



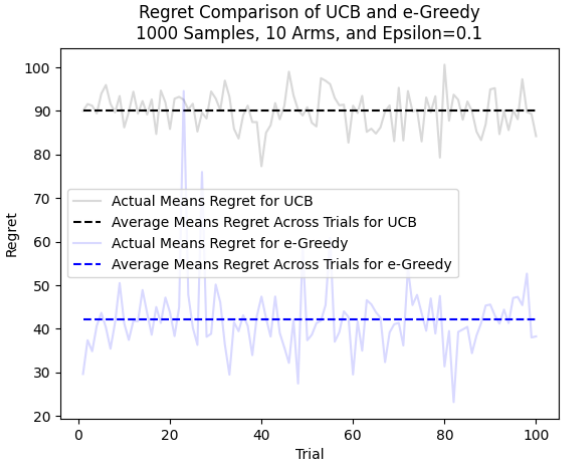




**1000 Samples, 100 Trials, 3 Arms, Epsilon=0.3 Std Dev=0.1**

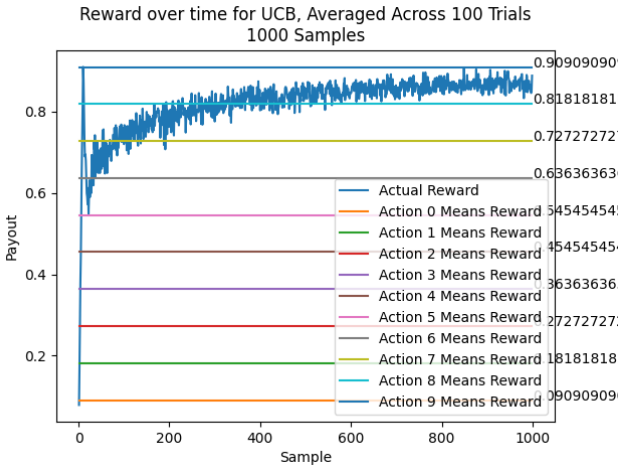
### 6.1.6 1000 Samples, 100 Trials, 10 Arms, Std Dev=0.1

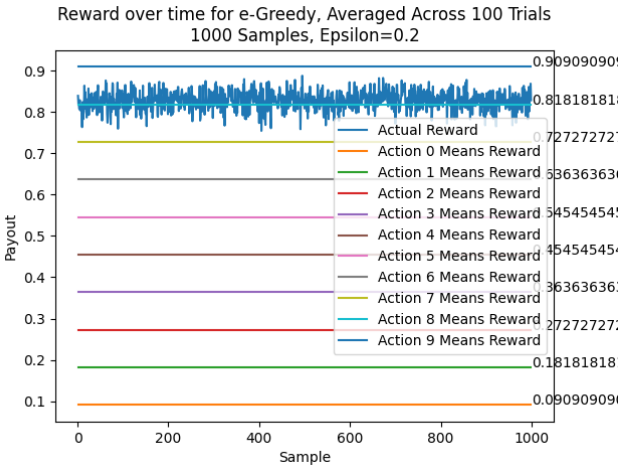


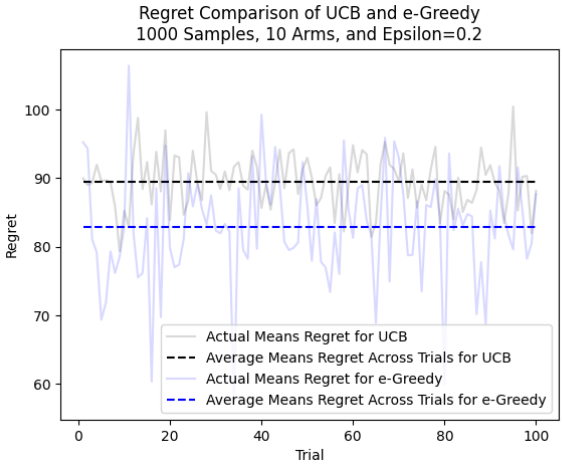



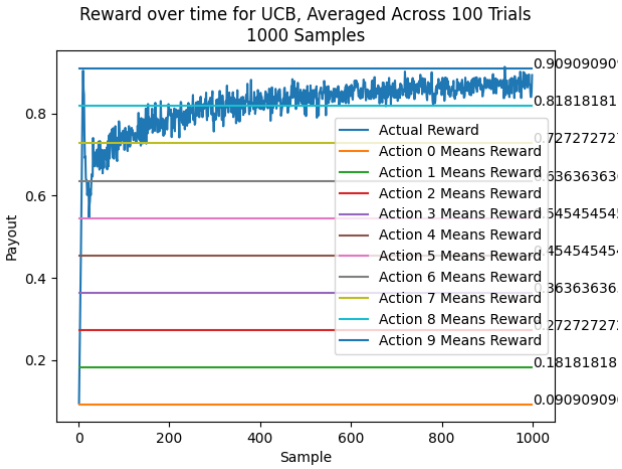

**1000 Samples, 100 Trials, 10 Arms, Epsilon=0.1 Std Dev=0.1**

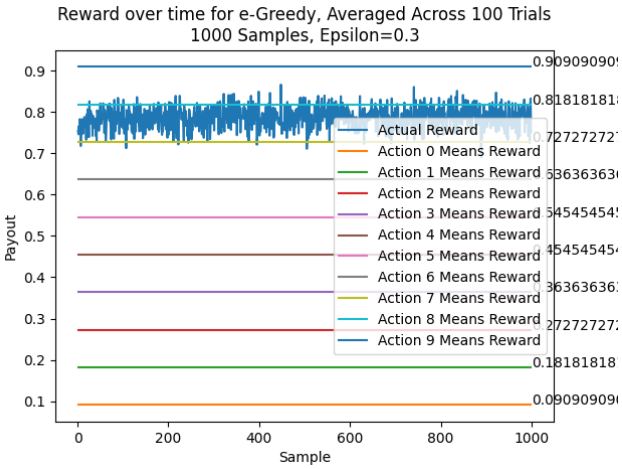


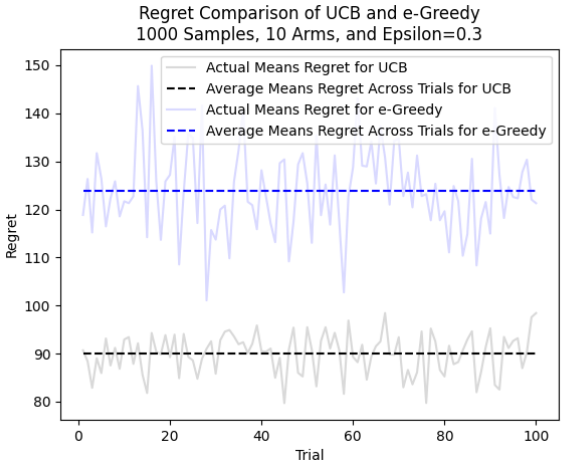




**1000 Samples, 100 Trials, 10 Arms, Epsilon=0.2 Std Dev=0.1**

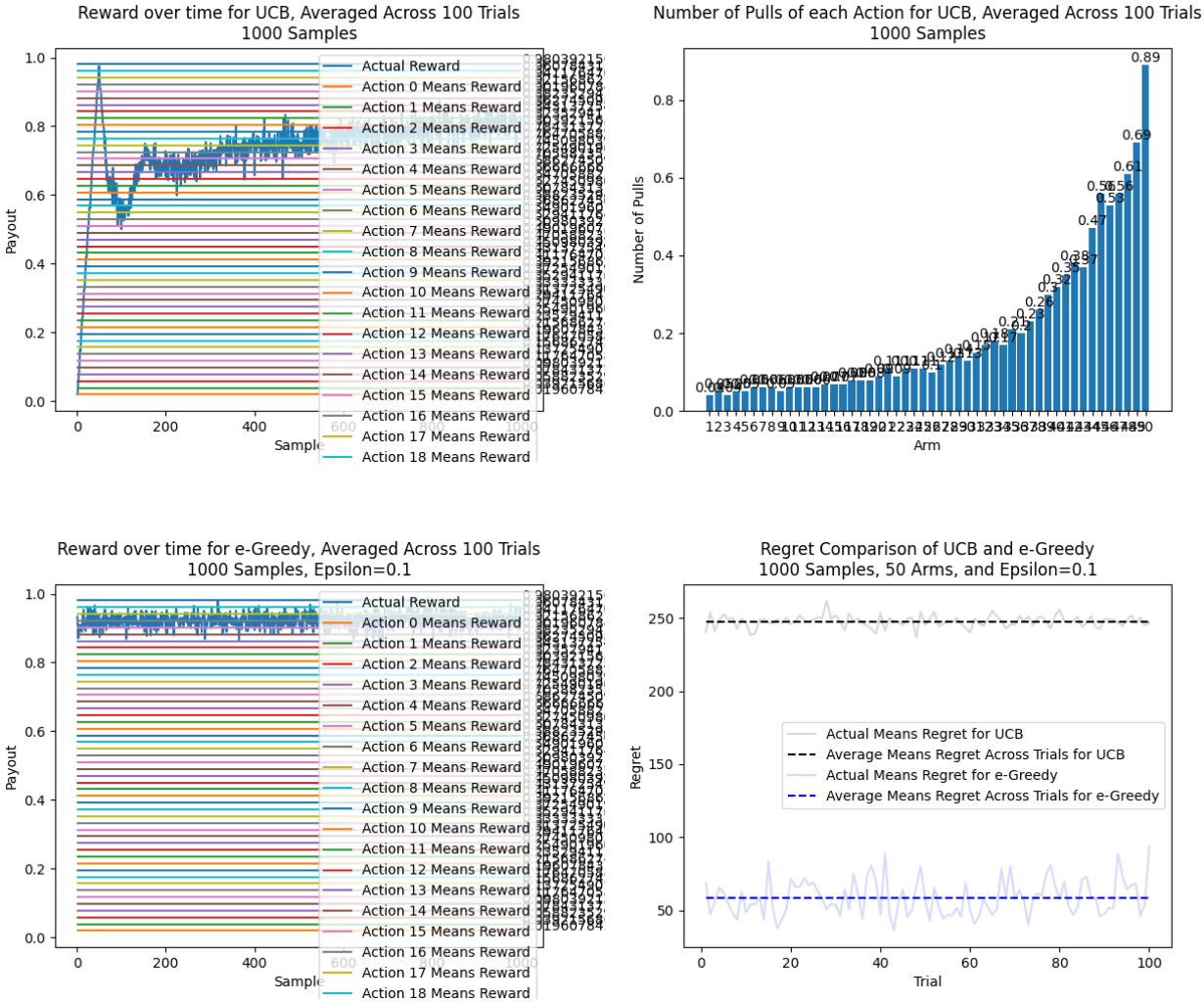


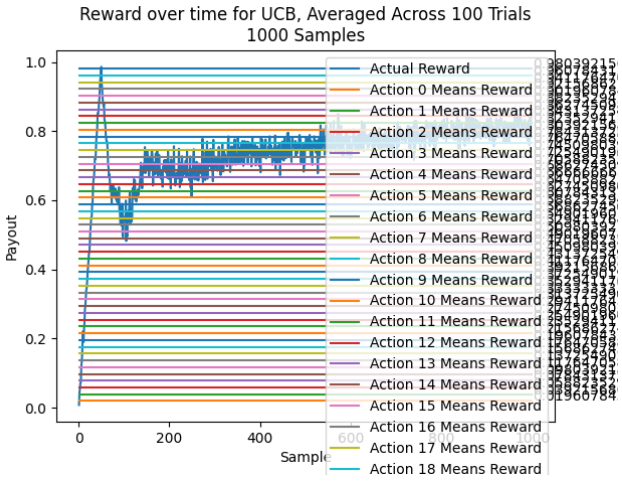



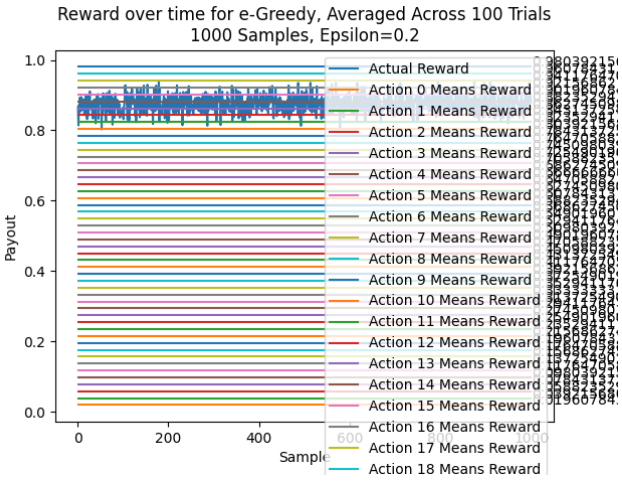

**1000 Samples, 100 Trials, 10 Arms, Epsilon=0.3 Std Dev=0.1**

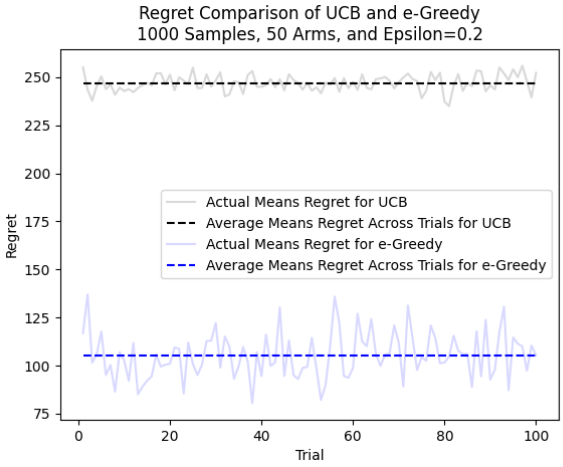
### 6.1.7 1000 Samples, 100 Trials, 50 Arms, Std Dev=0.1



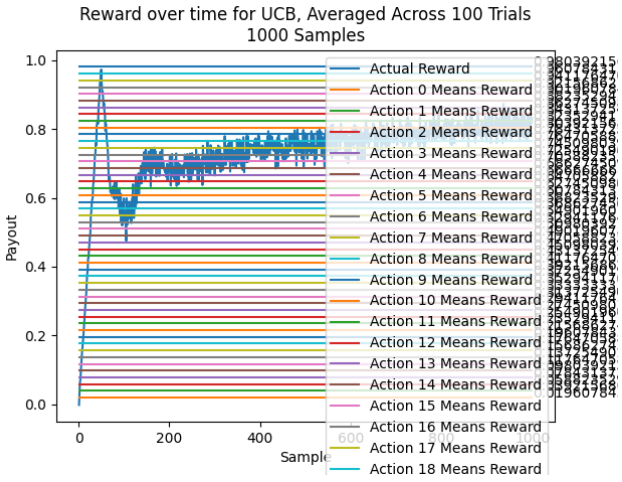

**1000 Samples, 100 Trials, 50 Arms, Epsilon=0.1 Std Dev=0.1**



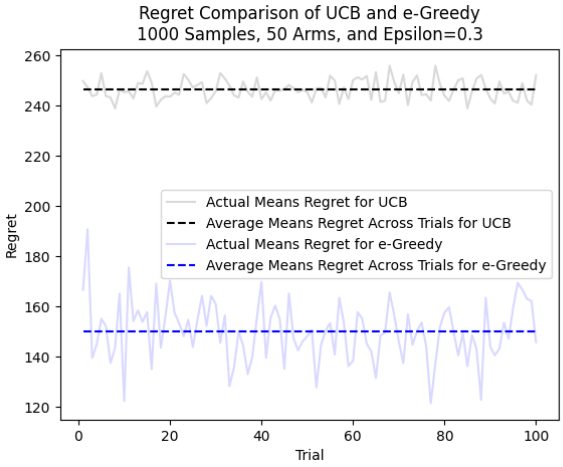




**1000 Samples, 100 Trials, 50 Arms, Epsilon=0.2 Std Dev=0.1**

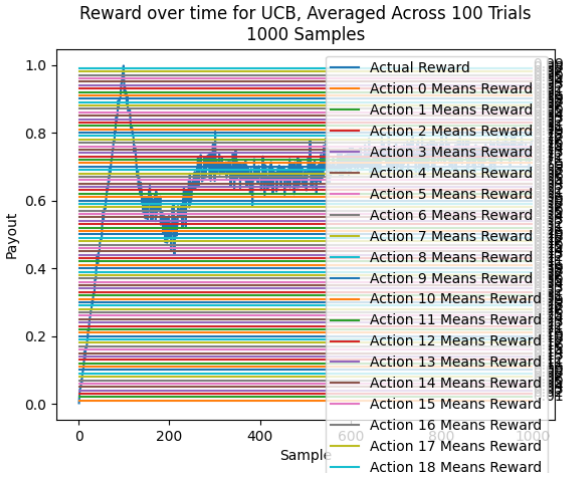


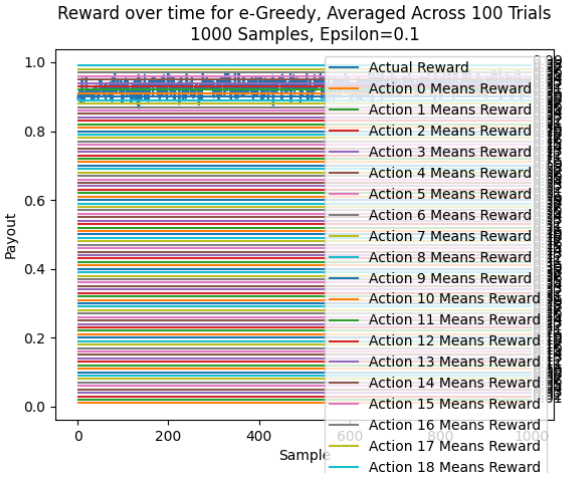


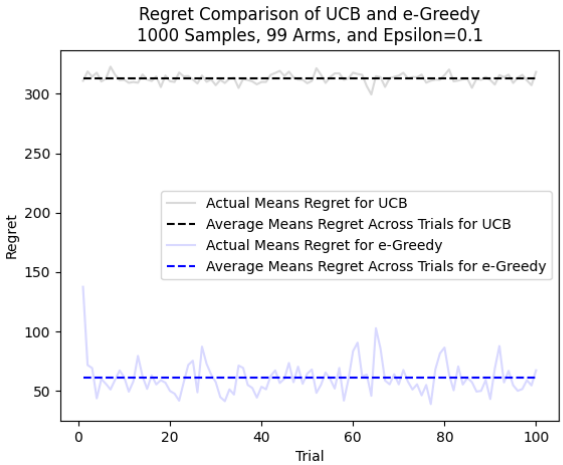



**1000 Samples, 100 Trials, 50 Arms, Epsilon=0.3 Std Dev=0.1**

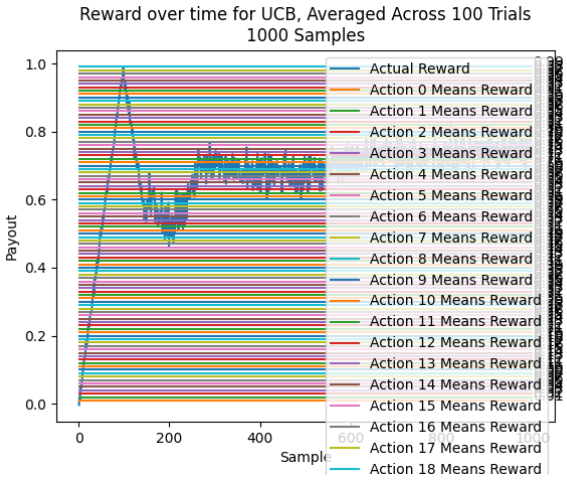
### 6.1.8 1000 Samples, 100 Trials, 99 Arms, Std Dev=0.1

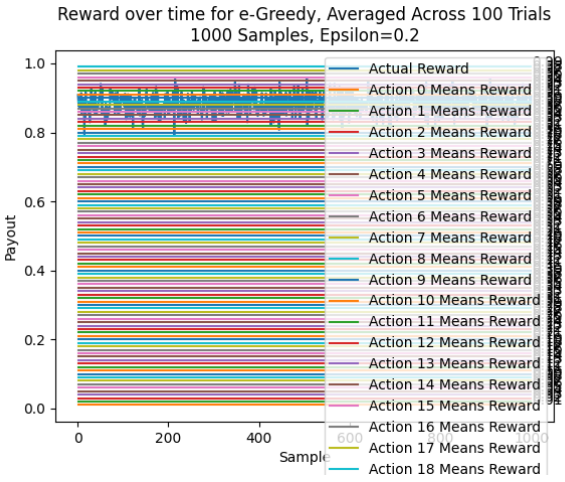


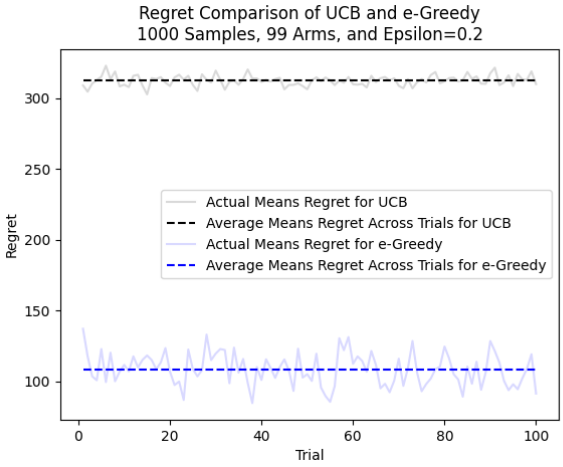



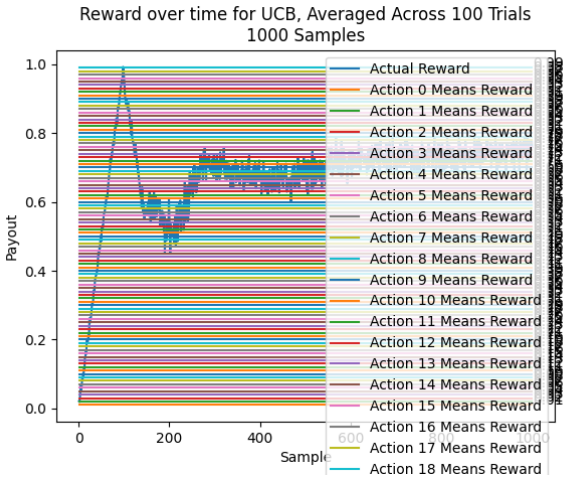

**1000 Samples, 100 Trials, 99 Arms, Epsilon=0.1 Std Dev=0.1**

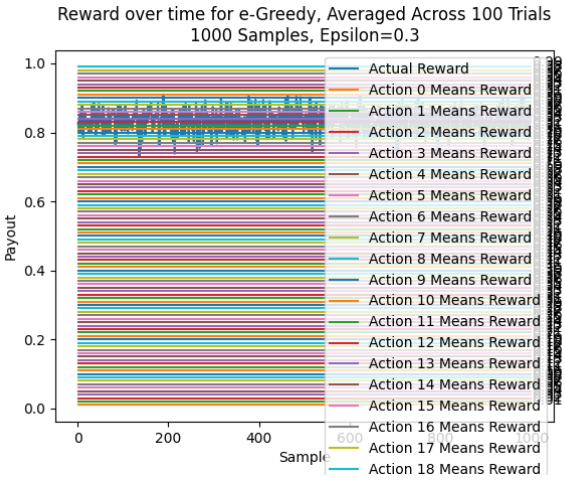


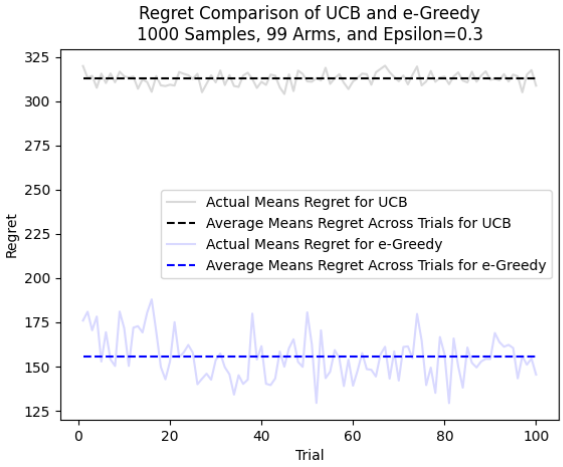




**1000 Samples, 100 Trials, 99 Arms, Epsilon=0.2 Std Dev=0.1**



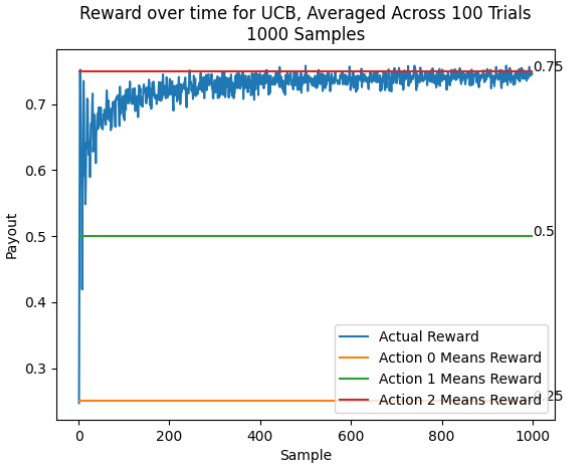



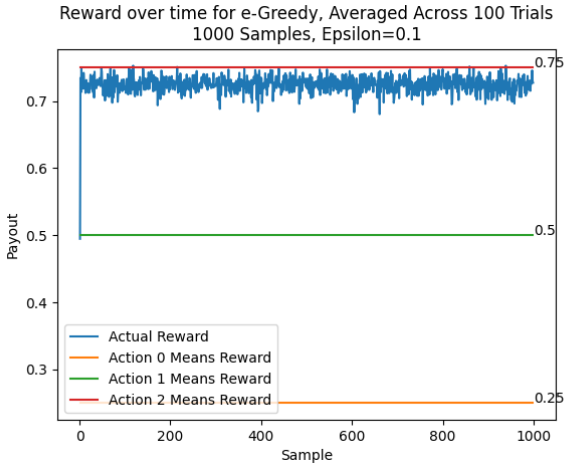



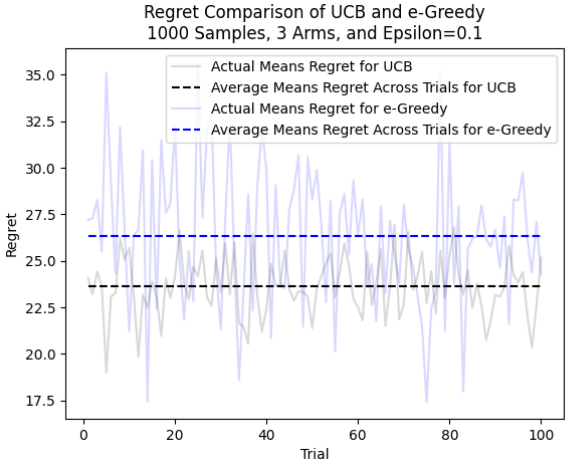

**1000 Samples, 100 Trials, 99 Arms, Epsilon=0.3 Std Dev=0.1**

## 6.2 Additional Domain: Standard Deviation = 0.05

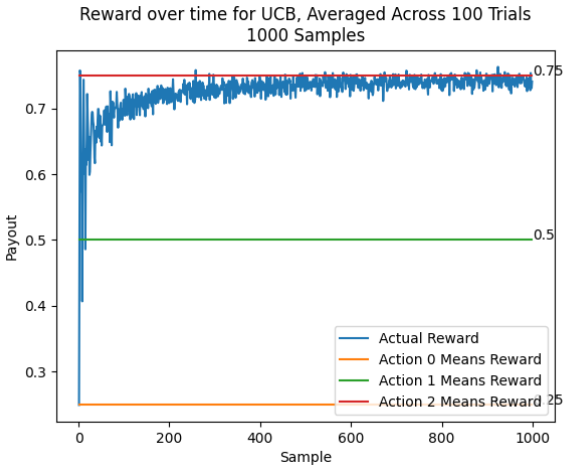
### 6.2.1 1000 Samples, 100 Trials, 3 Arms, Std Dev=0.05

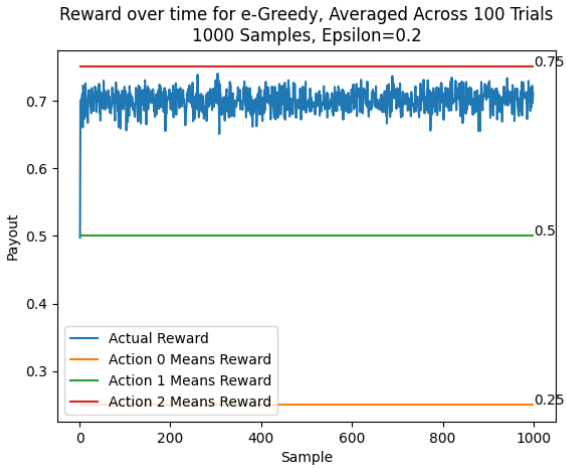


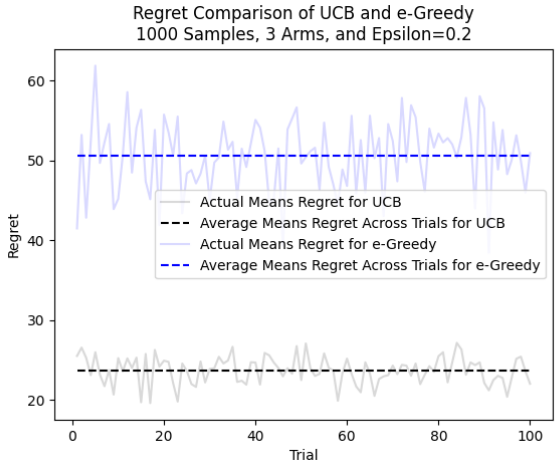



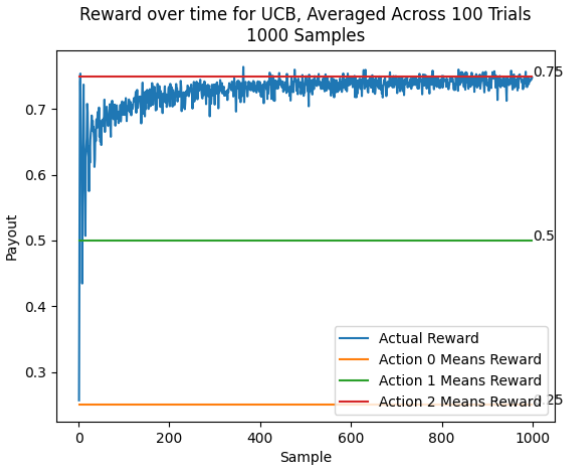

**1000 Samples, 100 Trials, 3 Arms, Epsilon=0.1 Std Dev=0.05**

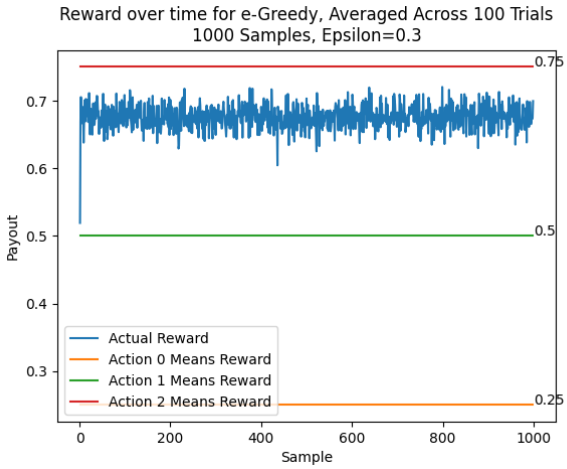


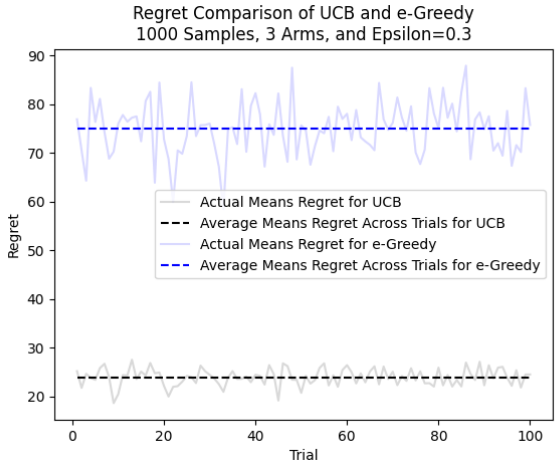




**1000 Samples, 100 Trials, 3 Arms, Epsilon=0.2 Std Dev=0.05**

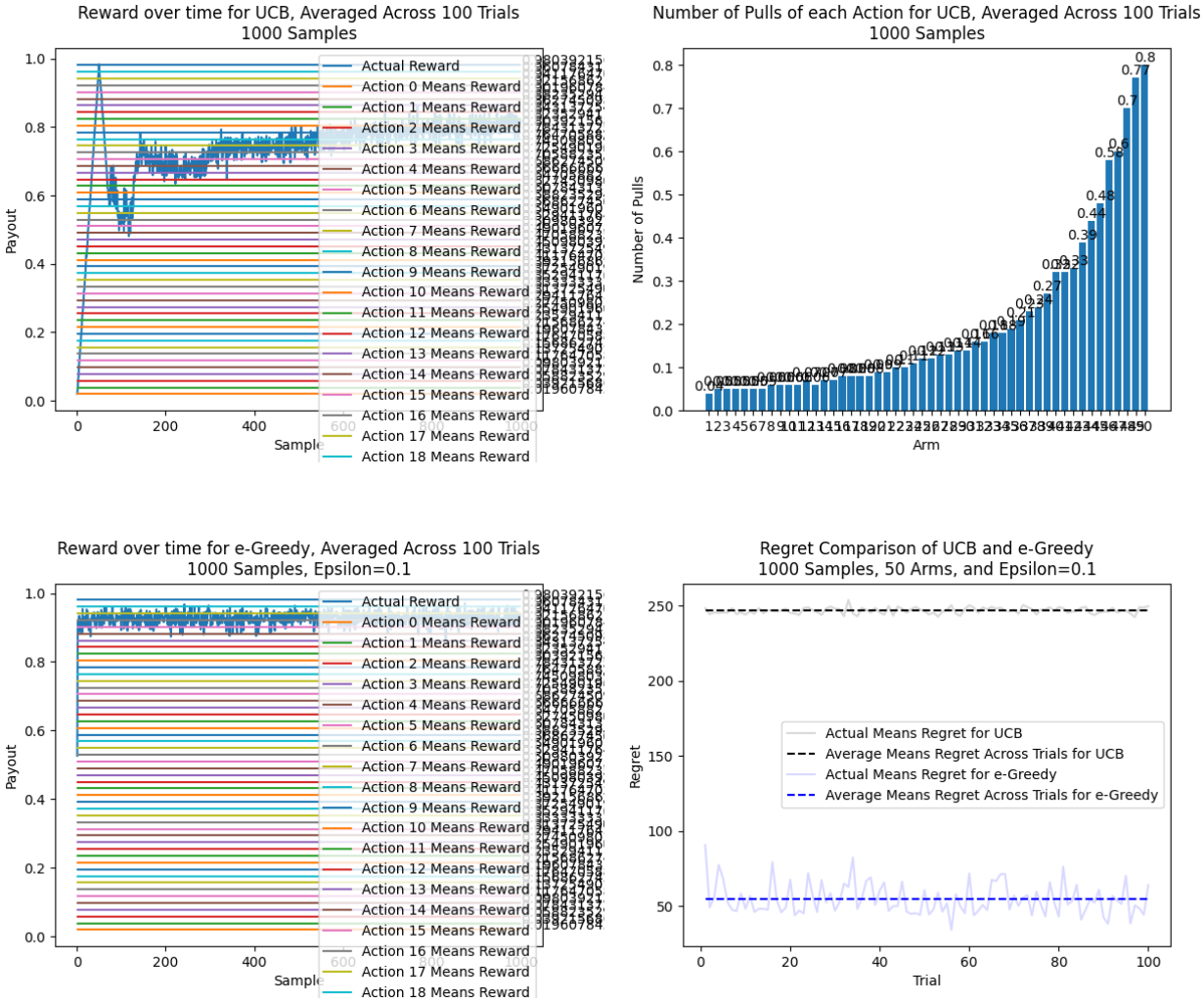


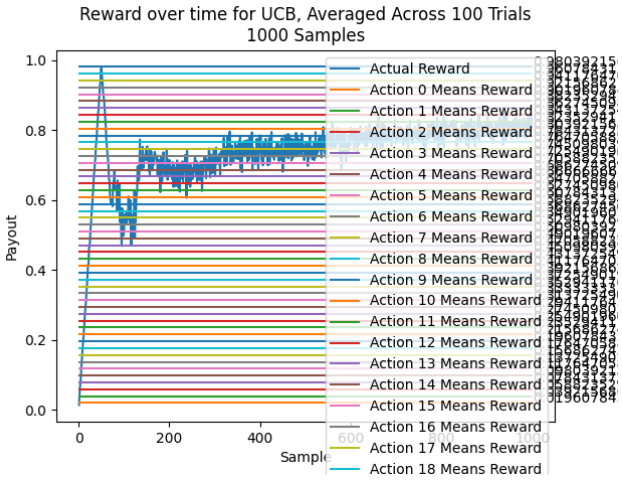



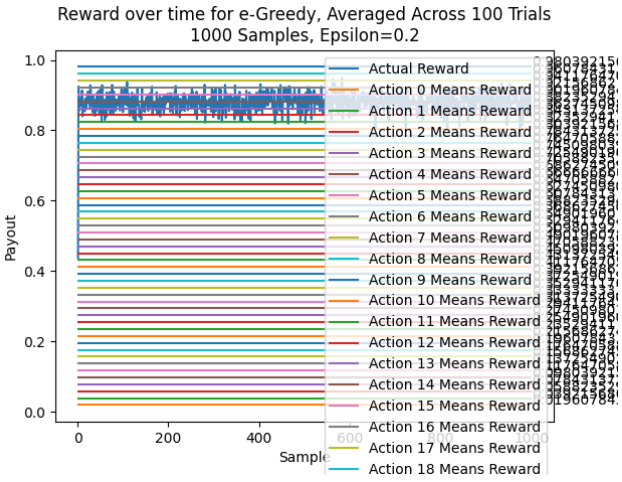

**1000 Samples, 100 Trials, 3 Arms, Epsilon=0.3 Std Dev=0.05**

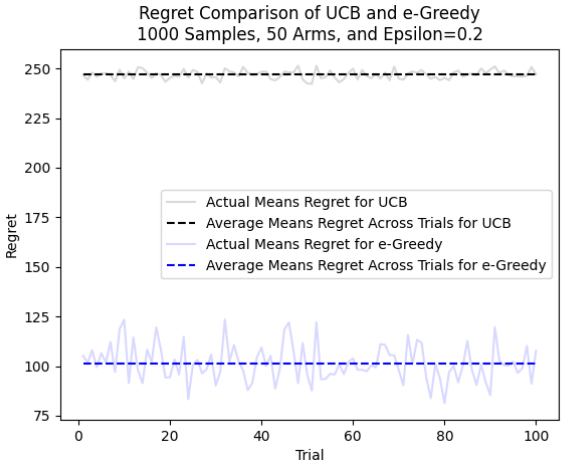
### 6.2.2 1000 Samples, 100 Trials, 50 Arms, Std Dev=0.05



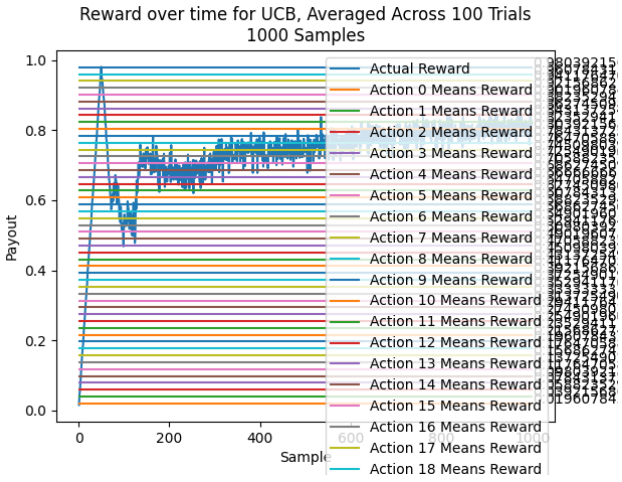

**1000 Samples, 100 Trials, 50 Arms, Epsilon=0.1 Std Dev=0.05**

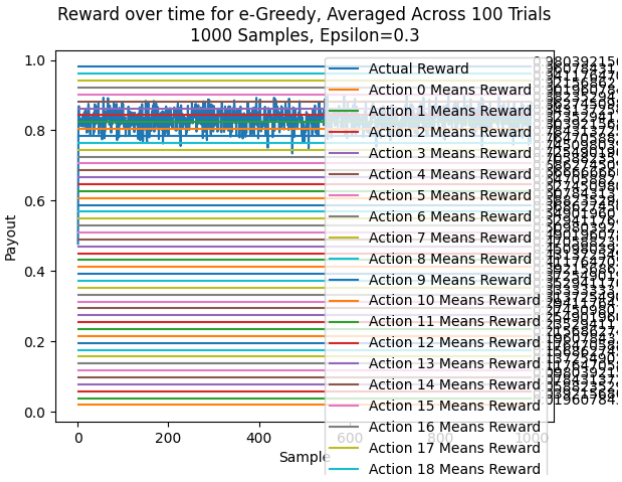


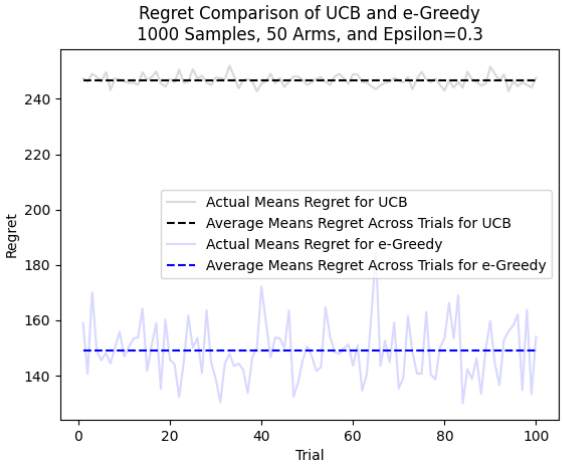




**1000 Samples, 100 Trials, 50 Arms, Epsilon=0.2 Std Dev=0.05**

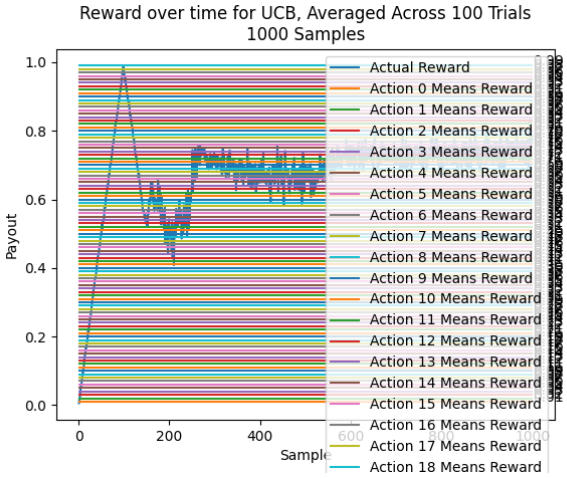


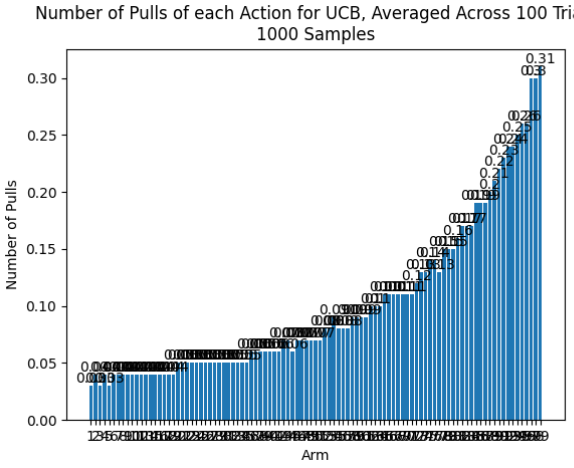



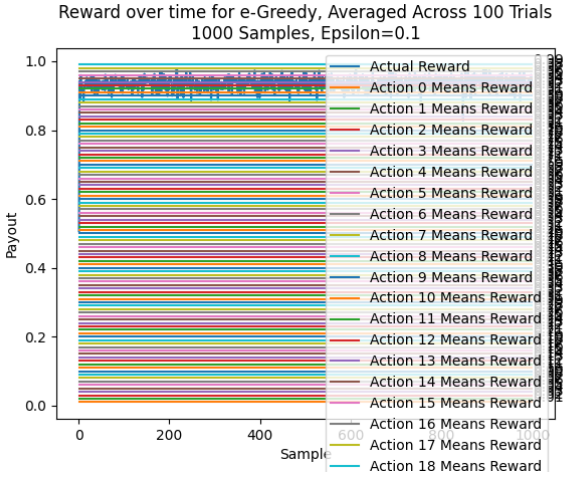



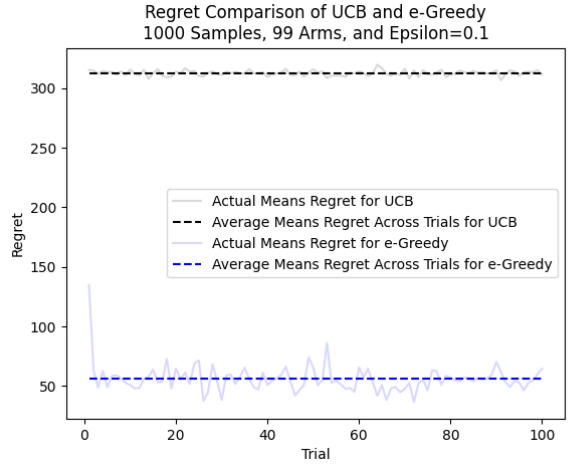

**1000 Samples, 100 Trials, 50 Arms, Epsilon=0.3 Std Dev=0.05**

### 6.2.3 1000 Samples, 100 Trials, 99 Arms, Std Dev=0.05

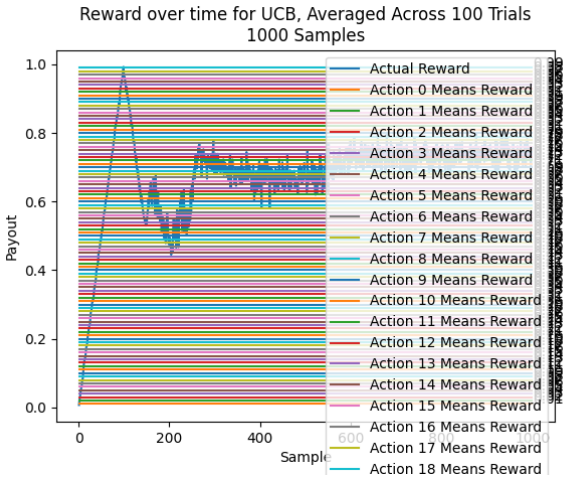


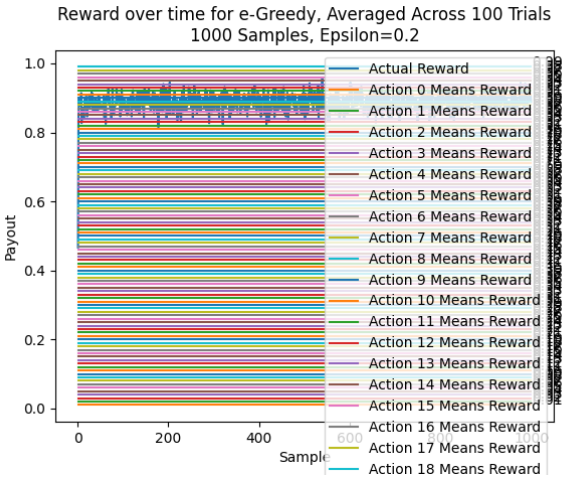



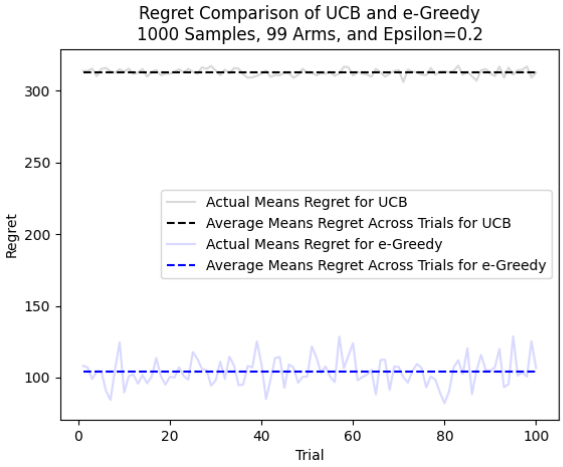



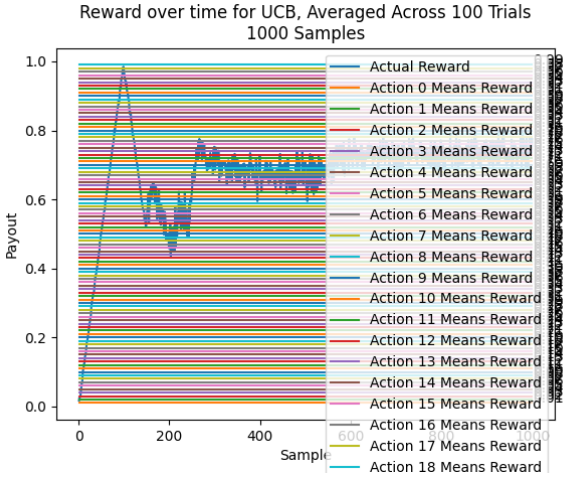

**1000 Samples, 100 Trials, 99 Arms, Epsilon=0.1 Std Dev=0.05**

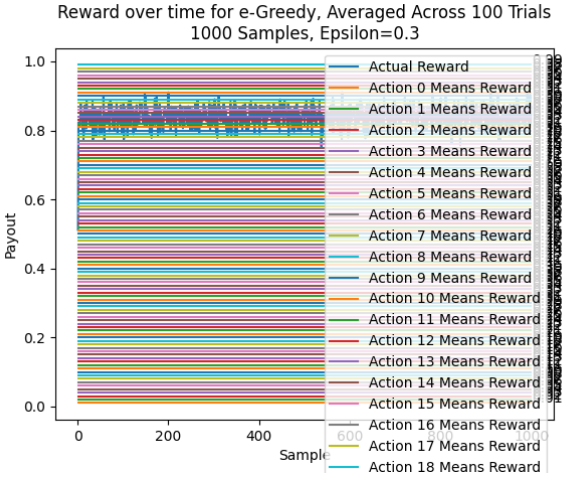


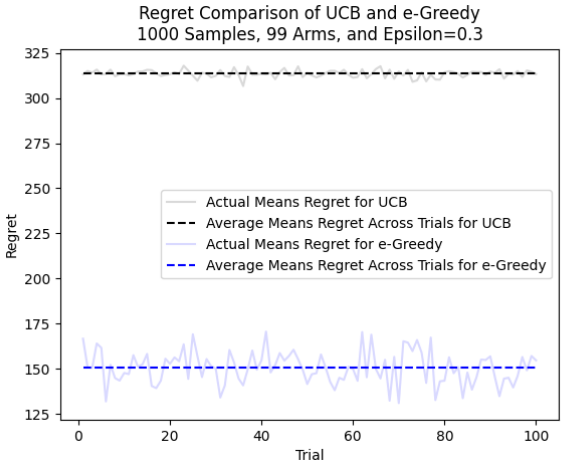




**1000 Samples, 100 Trials, 99 Arms, Epsilon=0.2 Std Dev=0.05**



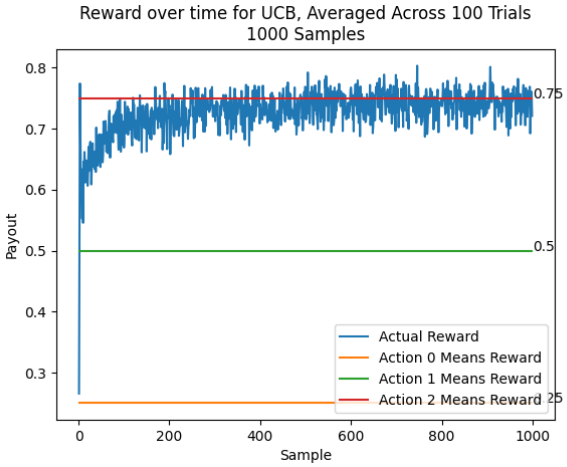



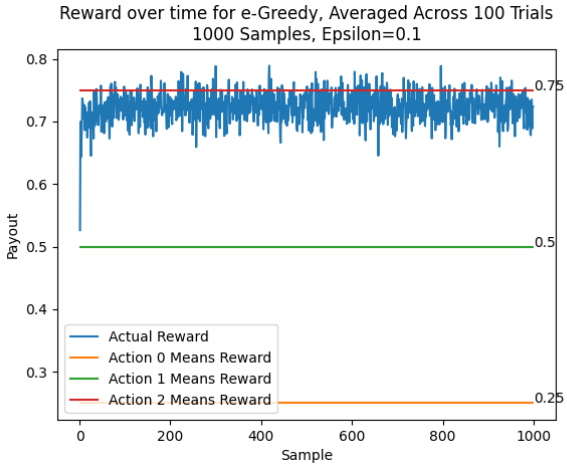



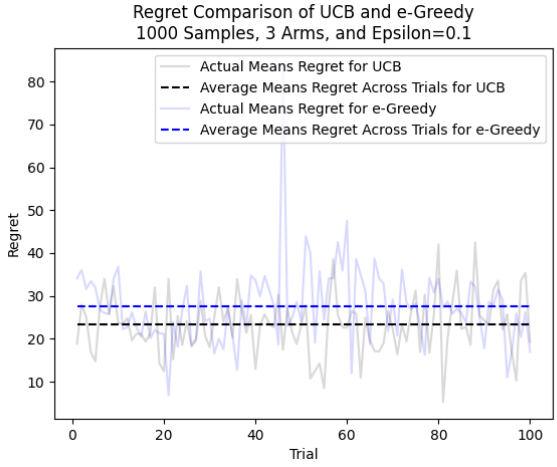

**1000 Samples, 100 Trials, 99 Arms, Epsilon=0.3 Std Dev=0.05**

## 6.3 Additional Domain: Standard Deviation = 0.2

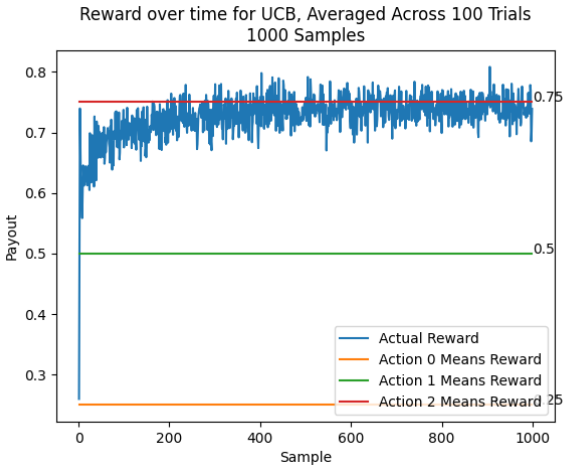
### 6.3.1 1000 Samples, 100 Trials, 3 Arms, Std Dev=0.2

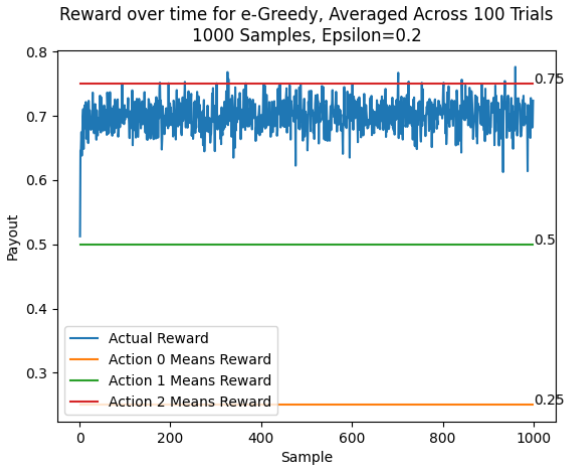


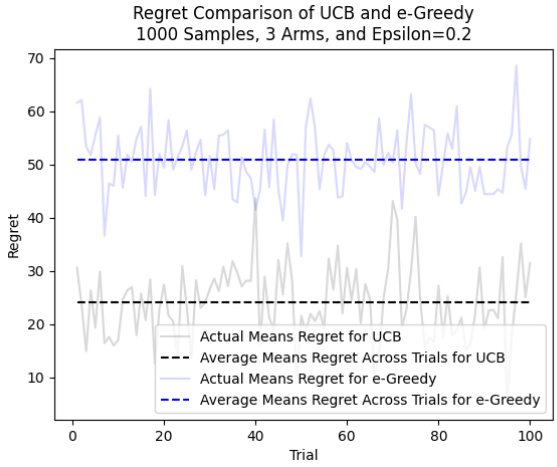



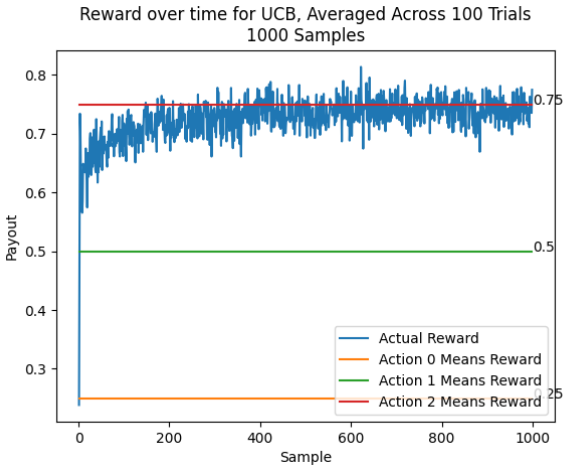

**1000 Samples, 100 Trials, 3 Arms, Epsilon=0.1 Std Dev=0.2**

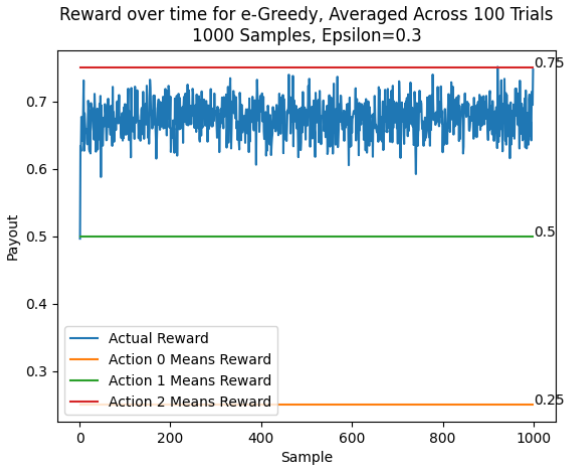


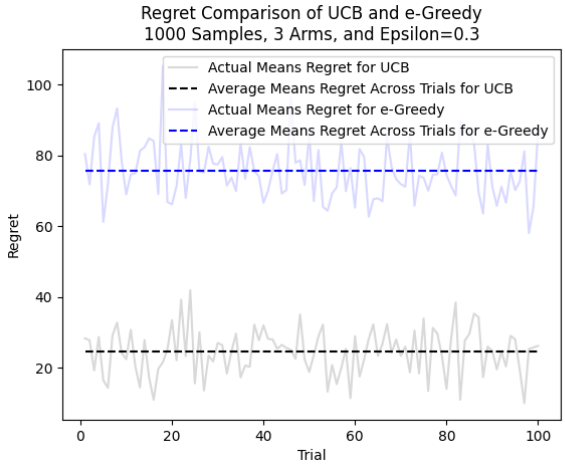




**1000 Samples, 100 Trials, 3 Arms, Epsilon=0.2 Std Dev=0.2**

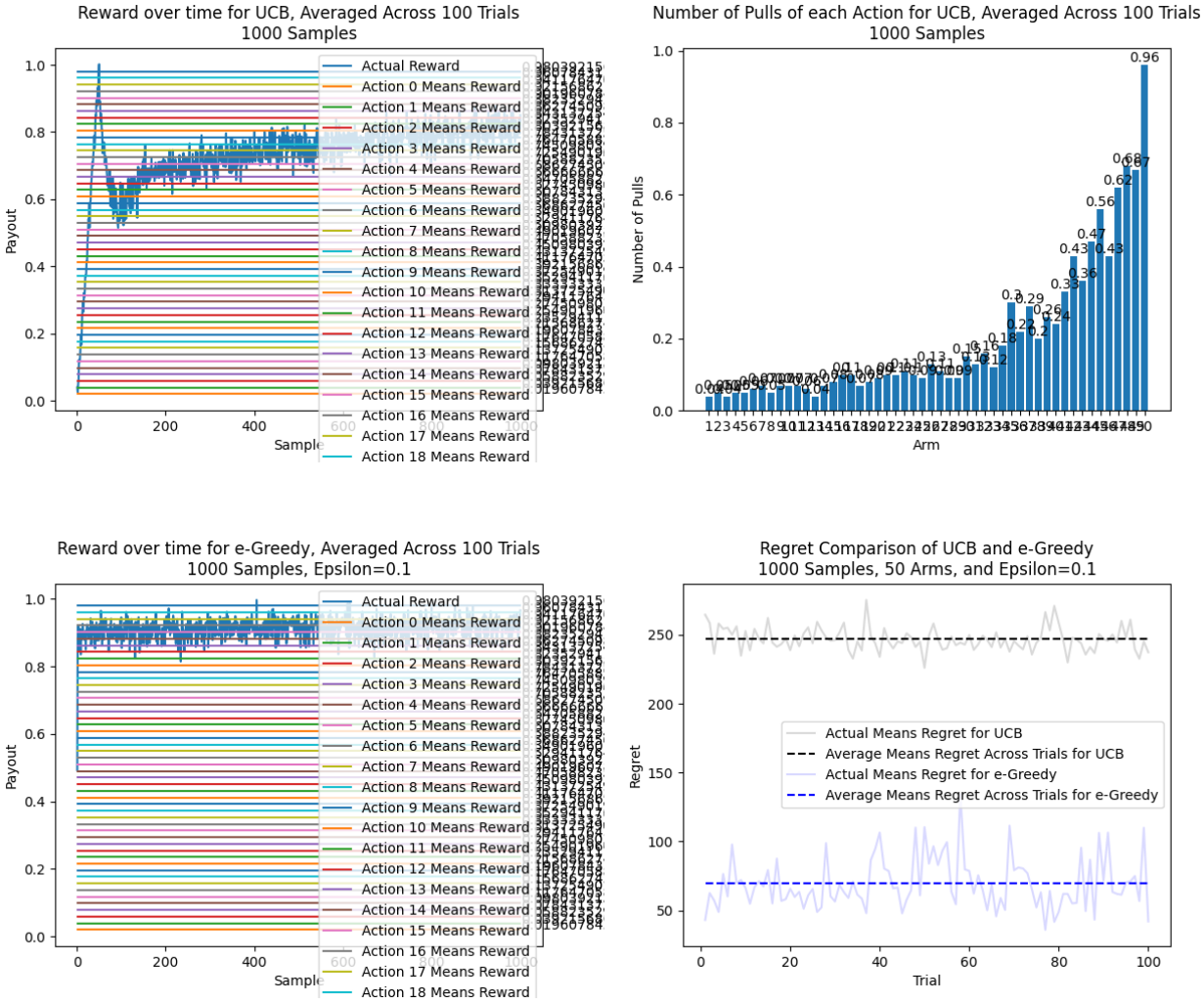


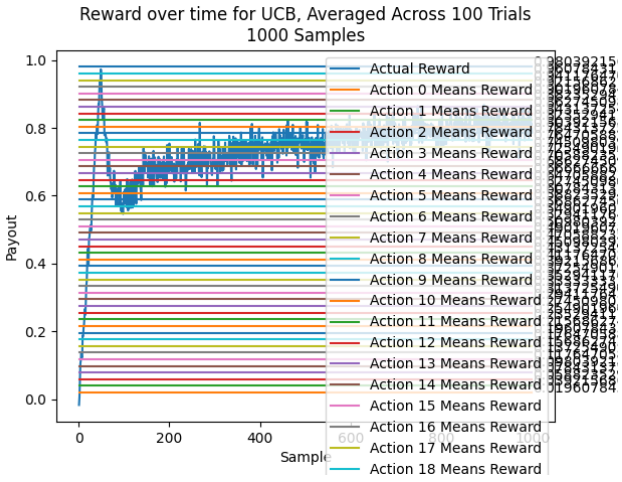



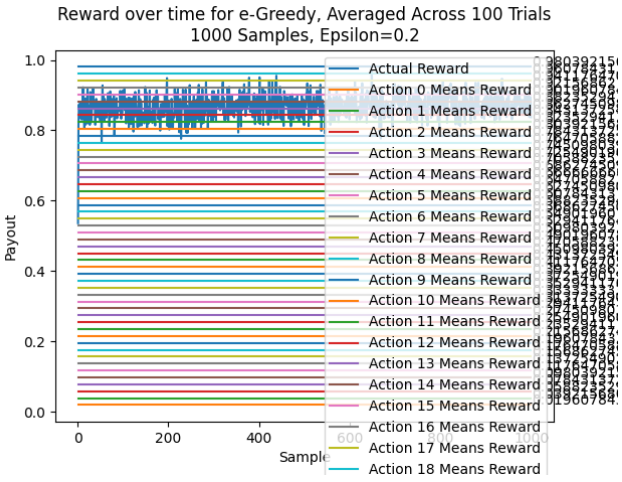

**1000 Samples, 100 Trials, 3 Arms, Epsilon=0.3 Std Dev=0.2**

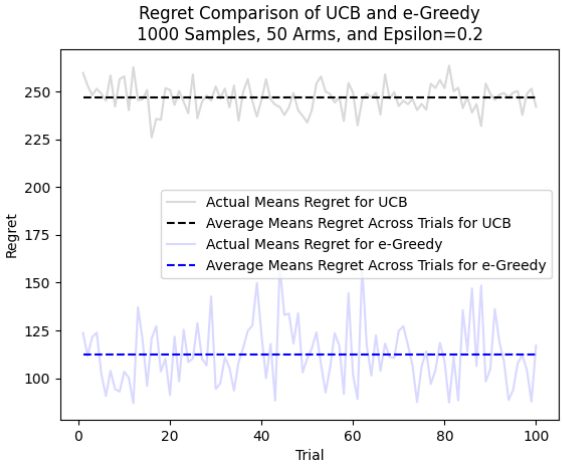
### 6.3.2 1000 Samples, 100 Trials, 50 Arms, Std Dev=0.2



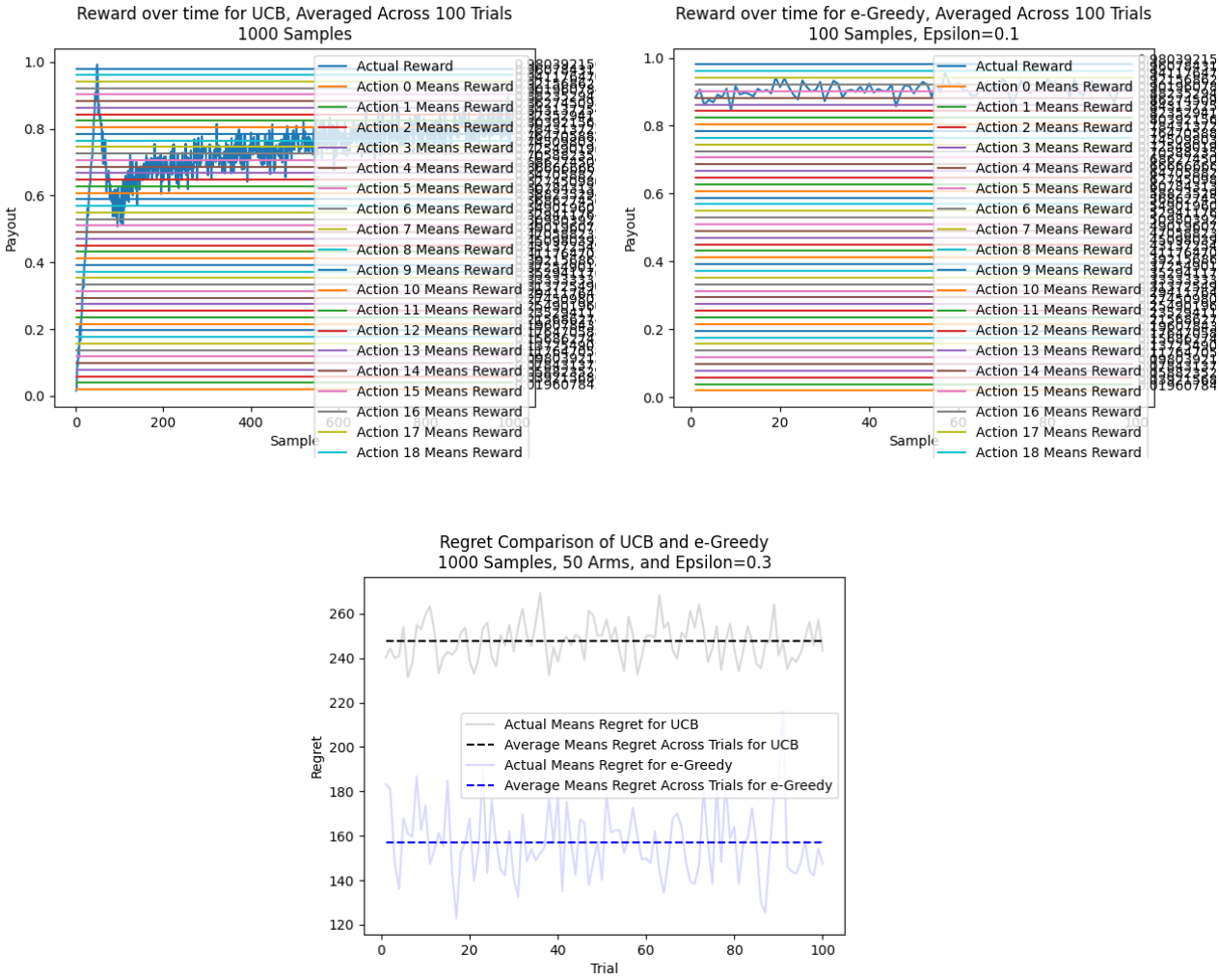

**1000 Samples, 100 Trials, 50 Arms, Epsilon=0.1 Std Dev=0.2**



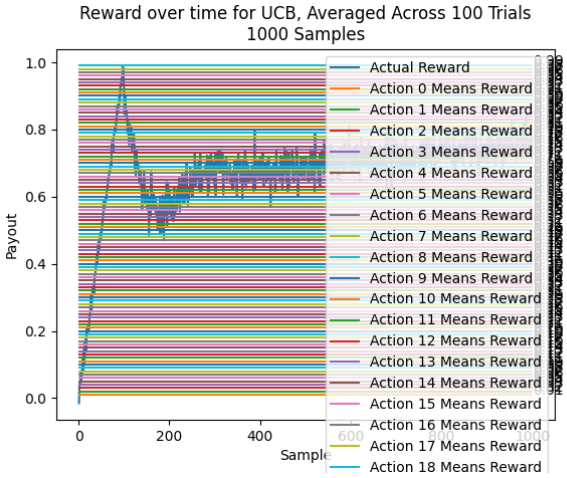



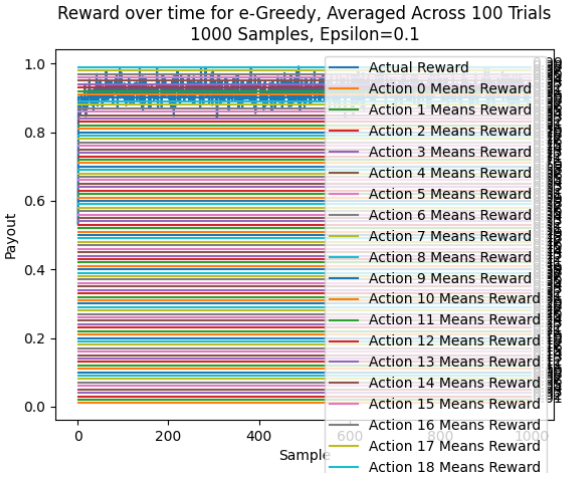

**1000 Samples, 100 Trials, 50 Arms, Epsilon=0.2 Std Dev=0.2**

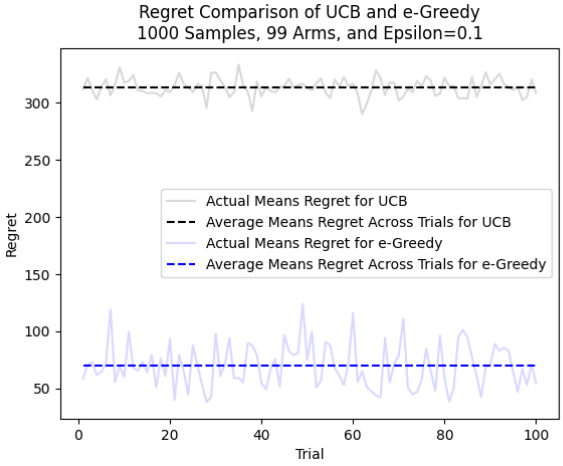



**1000 Samples, 100 Trials, 50 Arms, Epsilon=0.3 Std Dev=0.2**

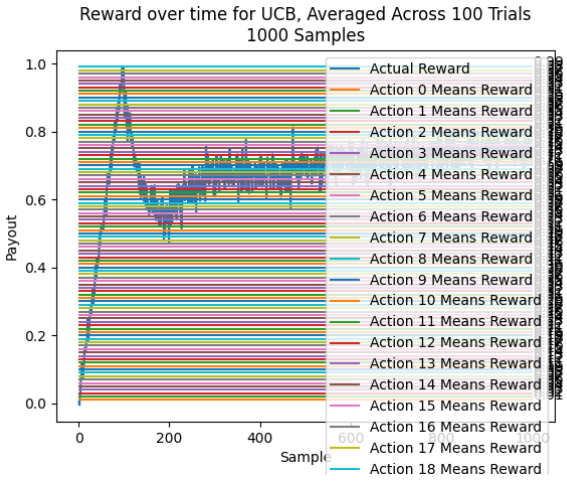
### 6.3.3 1000 Samples, 100 Trials, 99 Arms, Std Dev=0.2



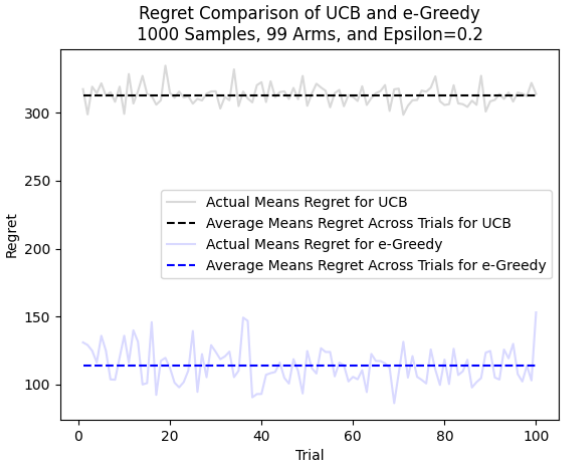



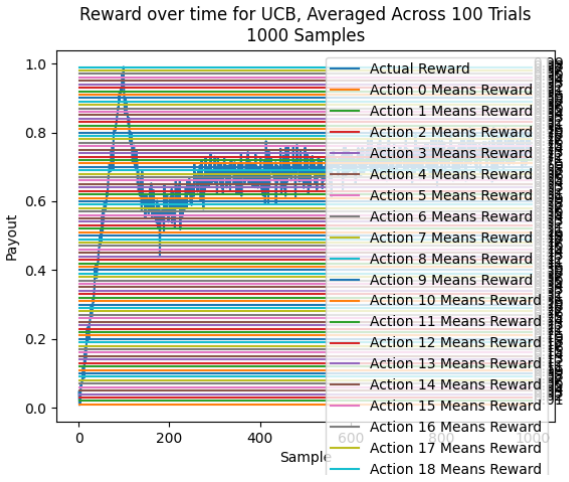

**1000 Samples, 100 Trials, 99 Arms, Epsilon=0.1 Std Dev=0.2**



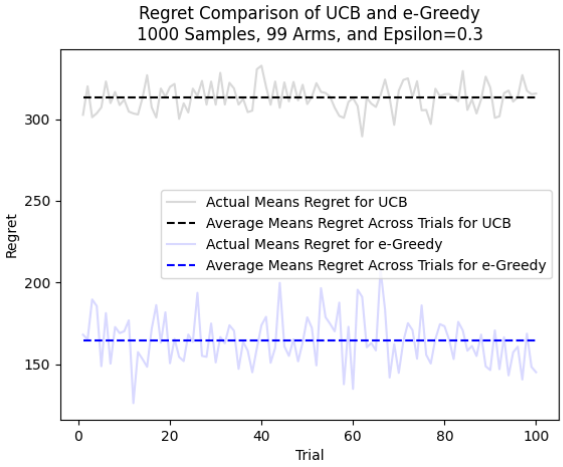





**1000 Samples, 100 Trials, 99 Arms, Epsilon=0.2 Std Dev=0.2**








**1000 Samples, 100 Trials, 99 Arms, Epsilon=0.3 Std Dev=0.2**