Exercises - Chapter 2

Victoria Nagorski

Exercise 2.1

In ε-greedy action selection, for the case of the two actions and $\varepsilon = 0.5$, what is the probability that the greedy action is selected?

Since $\varepsilon = 0.5$, the algorithm will choose randomly 50% of the time. For the random choices, there will be a 50/50 chance of the greedy or non-greedy action. This would mean that there is a 75% chance of selection the greedy action.

Exercise 2.2

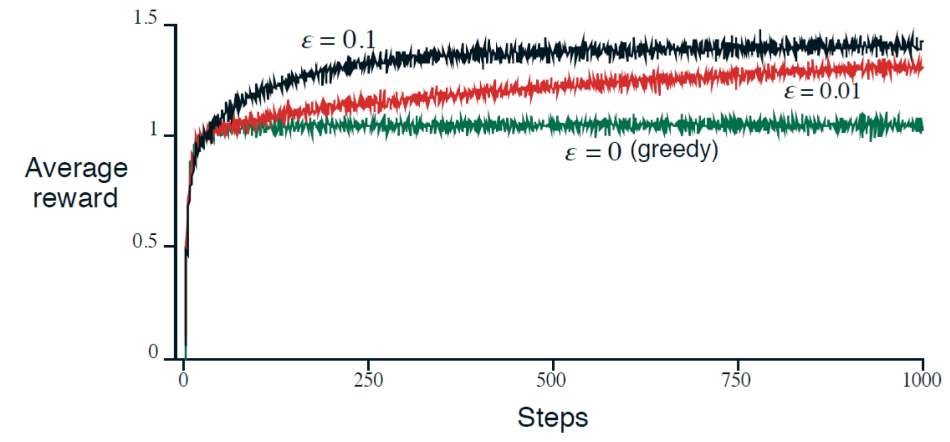
Consider a n-armed bandit problem with $n = 4$ actions, denoted 1,2,3, and 4. Consider applying to this problem a bandit algorithm using ε-greedy action selection, sample-average action-value estimates, and initial estimates of $Q_1(a) = 0$, for all a. Suppose the initial sequence of actions and rewards is $A_1 = 1$, $R_1 = 1$, $A_2 = 2$, $R_2 = 1$, $A_3 = 2$, $R_3 = 2$, $A_4 = 3$, $R_4 = 2$, $A_5 = 3$, $R_5 = 0$. On some of these time steps the ε case may have occurred, causing an action to be selected at random. On which time step did this definitely occur? On which time steps could this possibly have occurred?

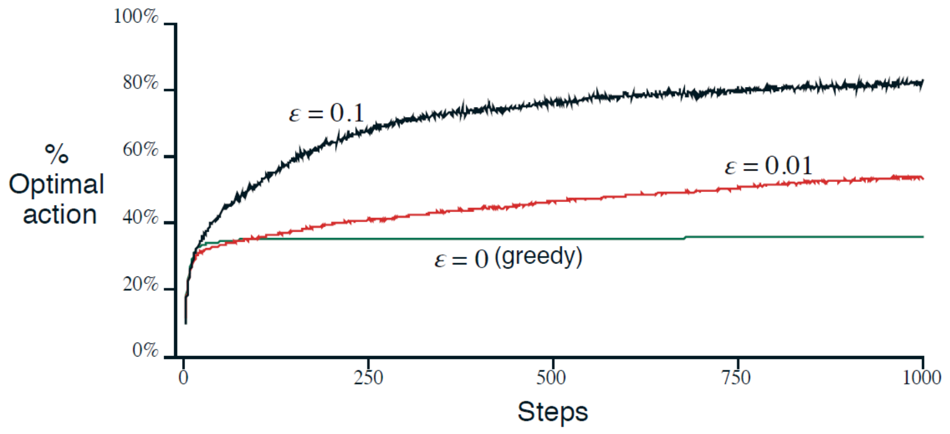
The random jump most definitely occurred at A_5 because the algorithm went from receiving a reward of 2 to a reward of 0 instead of 2. A greedy algorithm would never do this unless prompted to do so by ε.

A random jump could have occurred at A_2 because the algorithm changed its action. It is also possible for the algorithm to do a random action, and still do the optimal action. So really, any of the time steps could have been the random action.

Exercise 2.3

In the comparison shown in Figure 2.2, which method will perform best in the long run in terms of cumulative reward and cumulative probability of selecting the best action? How much better will it be? Express your answer quantitatively.





The $\varepsilon = 0.01$ will preform the best long term. $\varepsilon = 0.1$ will preform better at first because it will explore more aggressively and find the optimal choices faster. However, 10% of the time it is supposed to choose randomly. $\varepsilon = 0.01$ will take longer to determine the optimal choice, but will only choose randomly 1% of the time.

Exercise 2.4

If the step-size parameters, $\alpha_n(a)$, are not constant, then the estimate Q_n is a weighted average of previously received rewards with a weighting different from that given by (2.6). What is the weighting on each prior reward for the general case, analogous to (2.6), in terms of the sequence of step-size parameter? Equation 2.6:

$$Q_{n+1} = (1-\alpha)^n Q_1 + \sum_{i=1}^{n} \alpha(1-\alpha)^{n-i} R_i$$

If we let $\alpha = \frac{1}{n}$, the new equation becomes:

$$Q_{n+1} = (1-\frac{1}{n})^n Q_1 + \sum_{i=1}^{n} \frac{1}{n}(1-\frac{1}{n})^{n-i} R_i$$

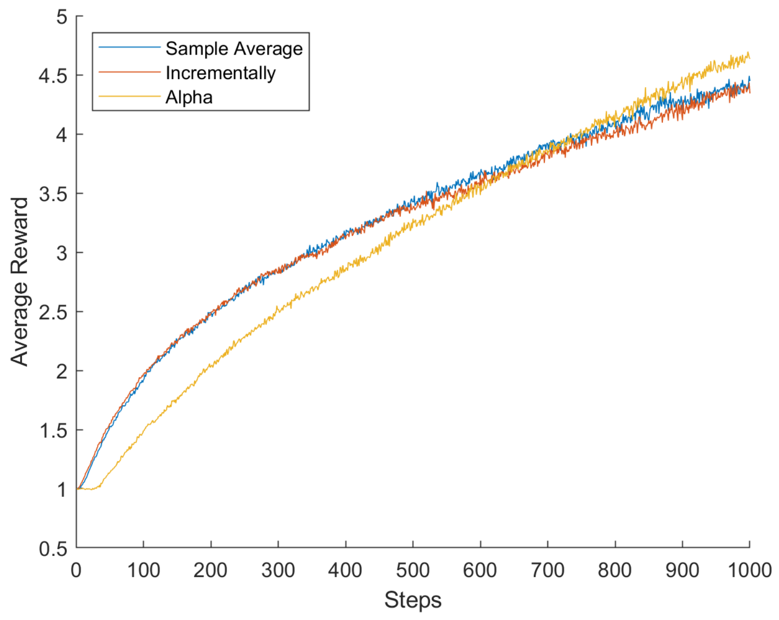
As the nth term increases, the term $\frac{1}{n}$ will become a really small number. This means that the $(1-\frac{1}{n})$ term will start off as a small number and then grow as the nth term increases.

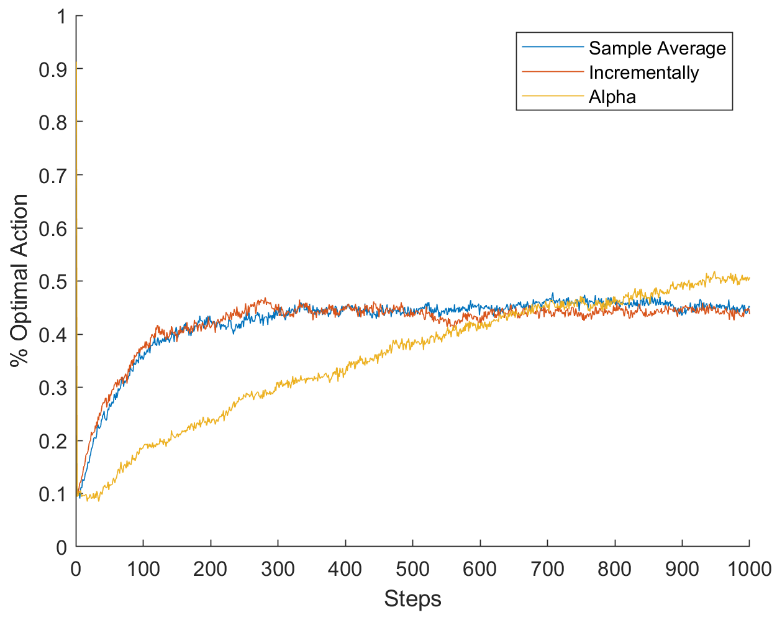
For the $(1-\frac{1}{n})^n Q_1$ term, this means it will experience less weight in the beginning because of the small size of $(1-\frac{1}{n})$, and then eventually continue to decrease because $(1-\frac{1}{n})$ is less than 1 and is to the nth power.

Looking at the summation term, the $(1-\frac{1}{n})^{n-i} R_i$ part of the equation will experience similar properties as the $(1-\frac{1}{n})^n Q_1$ term. However, only the first reward will experience an exponential decay to the nth term. The later rewards will have more weight because the exponential is $n-i$. The $\frac{1}{n}$ part of the summation removes more weight from each reward as the nth term increases through the simulation.

Exercise 2.5

Design and conduct an experiment to demonstrate the difficulties that sample-average methods have for non-stationary problems. Use a modified version of the 10-armed test-bed in which all the $q_\star(a)$ start out equal and then take independent random walks (say by adding a normally distributed increment with mean zero and standard deviation 0.01 to all the $q_\star(a)$ on each step). Prepare plots like Figure 2.2 for an action-value method using sample averages, incrementally computed, and another action-value method using a constant step-size parameter, $\alpha = 0.1$. Use $\varepsilon = 0.1$ and longer runs, say of 1,000 steps.



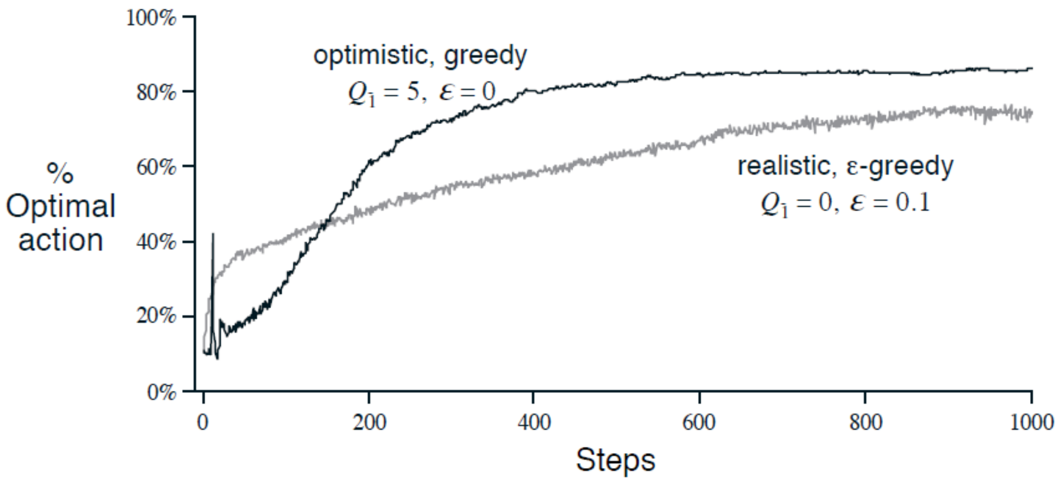


Listing 1: Main Script

```
% Initialize Variables
epsilon = 0.1; % Greedy factor
n = 10; % Number of arms
steps = 1000;
alpha = 0.1; % contant
std = [0,sqrt(0.01)]; % [Mean, Sigma]
runs = 2000; % Number of runs to average over
version = ["Average","Incremental","Const"];
location = 'moving';
type = 'version';

% Run Test-Bed
[final_rewards,final_actions] = k_bandit_testbed(epsilon,n,steps,version,↩
    location,std,type,runs,alpha);

% Plot Data
figure(1)
hold on
plot((1:steps),final_rewards(:,1))
plot((1:steps),final_rewards(:,2))
plot((1:steps),final_rewards(:,3))
legend('Sample Average','Incrementally','Alpha')
xlabel('Steps')
ylabel('Average Reward')
hold off

figure(2)
hold on
plot((1:steps),final_actions(:,1))
plot((1:steps),final_actions(:,2))
plot((1:steps),final_actions(:,3))
legend('Sample Average','Incrementally','Alpha')
xlabel('Steps')
ylabel('% Optimal Action')
hold off
```

Exercise 2.6

The results shown in Figure 2.3 should be quite reliable because they are averages over 2000 individual, randomly chosen 10-armed bandit tasks. Why, then, are there oscillations and spikes in the early part of the curve for the optimistic method? In other words, what might make this method perform particularly better or worse, on average, on particular early steps?



The $q_\star(a)$ are selected from a normal distribution of mean 0 and variance of 1 - making the $+5$ start an extremely optimistic initial estimate. Consequently, the algorithm will act "disappointed" with the first rewards and do some exploration before beginning to converge. The spike most likely occurred because it discovered the optimal action, but was still disappointed compared the $+5$ start-off and continued to explore.

This method, on average, performs worse early on because of the explorative nature it exhibits in the beginning. Optimistic methods will exhibit exploratory tendencies even if they are greedy ($\varepsilon = 0$). The exploration of this algorithm will decrease with time, but this also makes this method non-optimal for non-stationary problems.

Exercise 2.7

In most of this chapter we have used sample averages to estimate action values because sample averages do not produce the initial bias that constant step sizes do (see the analysis in (2.6)). However, sample averages are not a completely satisfactory solution because they may perform poorly on non-stationary problems. Is it possible to avoid the bias of constant step sizes while retaining their advantages on non stationary problems? One way is to use a step size of $\beta_n \doteq \alpha/\bar{o}$ to process the nth reward for a particular action, where $\alpha > 0$ is a conventional constant step size, and $\bar{o}_n$ is a trace of one starts at 0:

$$\bar{o}_n \doteq \bar{o}_{n-1} + \alpha(1 - \bar{o}_{n-1}), \text{ for } n \geq 0, \text{ with } \bar{o}_0 \doteq 0$$

Carry out an analysis like that in (2.6) to show that Q_n is an exponential recency-weighted average without initial basis.

$$Q_{n+1} = (1-\alpha)^n Q_1 + \sum_{i=1}^{n} \alpha(1-\alpha)^{n-i} R_i$$

Substituting $\alpha = \beta_n$, the new equation becomes:

$$Q_{n+1} = (1-\beta_n)^n Q_1 + \sum_{i=1}^{n} \beta_n(1-\beta_n)^{n-i} R_i$$

Examining β_n, we find that this term will start off large and then decrease as the nth action is increased. Eventually $\bar{o}_n$ will go to 1, and you will be left with the a constant α.

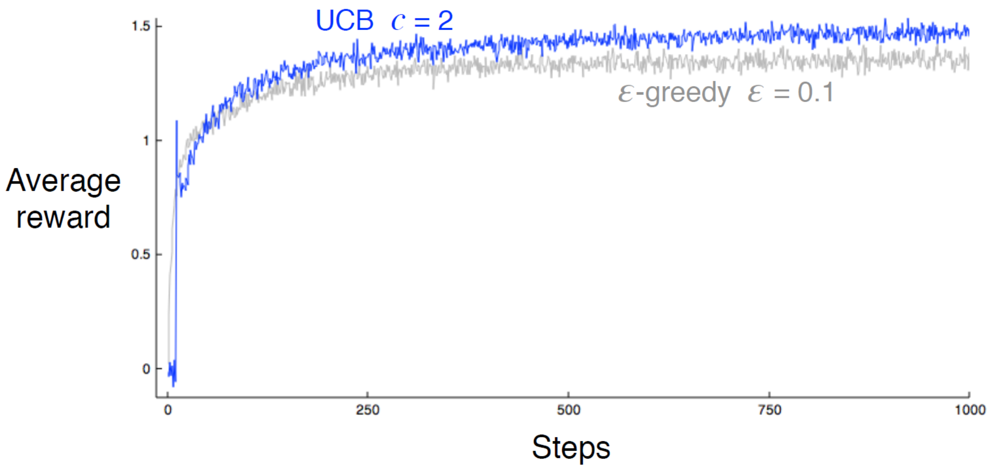
If you look at the first term, $(1-\beta_n)^n Q_1$, you will find that there is less weight added from this term in the very beginning. Eventually, the nth exponential will cause the term to decay exponentially to where Q_1 stops having any weight compared to more the recent moves.

Looking at the last term, $\sum_{i=1}^{n} \beta_n(1-\beta_n)^{n-i} R_i$, we will notice that the first terms experiences less weight than the later terms. A numerical example is when $n = 4$ and $\alpha = 0.1$. At R_1, the constant version will have reward weights in the following order $R_{1w} = 0.07, R_{2w} = 0.081,$ and $R_{3w} = 0.09$. However the β_n term will have the following weights per reward $R_{1w} = 0.1036, R_{2w} = 0.146,$ and $R_{3w} = 0.206$. The difference between R_{3w} and R_{1w} is much greater in the β_n example than the constant one. After a certain point, the nth exponential weight will begin to decay according to the constant alpha term as $\bar{o}_n$ goes to 1.

This is significant because the initial reward estimates factor less weight of earlier rewards into their calculation compared to the constant version- reducing initial basis. The reduction in the first initial biases of the expected rewards is important they effect greedy decisions made early on in the simulation.

Exercise 2.8

In Figure 2.4 the UCB algorithm shows a distinct spike in performance on the 11th step. Why is this? Note that your answer to be fully satisfactory it must explain both why the reward increases on the 11th step and why it decreases on the subsequent steps. Hint: if $c = 1$, then the spike is less prominent.



The following is the equation for UCB:

$$A_t \doteq \underset{a}{argmax}\left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}}\right]$$

The square-root part of this equation is the uncertainty in the estimate of the best action. Time is constantly increasing but will eventually start increasing at a decreasing rate. However, the $N_t(a)$ term is based on how many times an action has been chosen. Because of this, each action will theoretically be chosen at least once. The more times an action is chosen, the smaller square-root term will become. The c is a constant multiplier against this uncertainty term.

Assuming time to be the same for all actions, we do not need to worry about time in this analysis. The two main contributing factors to the uncertainty will be c and $N_t(a)$ at this point. The algorithm will then choose the best action, but now that action will have been chosen twice. Consequently the actions where $N_t(a) = 1$ multiplied by a greater c will create a greater uncertainty value to draw the algorithm to choose those actions instead of the optimal action (causing the sharp decrease).

Exercise 2.9

Show that in the case of two actions, the soft-max distribution is the same as that given by the logistic, or sigmoid, function often used in statistics and artificial neural networks.

The logistic function looks like:

$$S(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1}$$

The soft-max function is given to be:

$$Pr\{A_t = a\} \doteq \frac{e^{H_t(a)}}{\sum_{b=1}^{k} e^{H_t(b)}}$$

The soft-max is the relative preference of one action to another. Since we know that there is only two actions we can rewrite the soft-max equation in the following manner:

$$Pr\{a_1\} \doteq \frac{e^{H_t(a_1)}}{e^{H_t(a_1)} + e^{H_t(a_2)}}$$

$$Pr\{a_1\} \doteq \frac{e^{H_t(a_1)}}{e^{H_t(a_1)}\left(1 + \frac{e^{H_t(a_2)}}{e^{H_t(a_1)}}\right)} = \frac{e^{H_t(a_1)}}{e^{H_t(a_1)}\left(1 + e^{H_t(a_2)-H_t(a_1)}\right)}$$

Cancel terms:

$$Pr\{a_1\} \doteq \frac{\cancel{e^{H_t(a_1)}}}{\cancel{e^{H_t(a_1)}}\left(1 + e^{H_t(a_2)-H_t(a_1)}\right)} = \frac{1}{\left(1 + e^{H_t(a_2)-H_t(a_1)}\right)}$$

If we set $-x = H_t(a_2) - H_t(a_1)$, then the two functions would be equivalent.

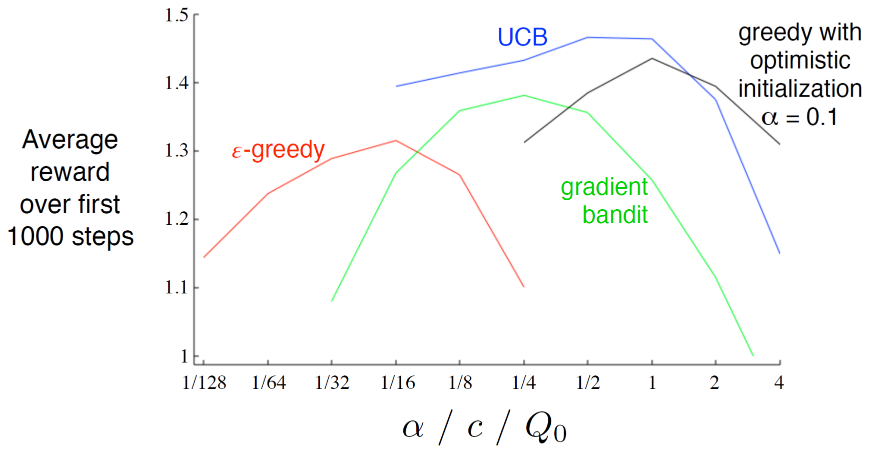
Exercise 2.10

Suppose you face a binary bandit task whose true action values change randomly from play to play. Specifically, suppose that for any play the true values of actions 1 and 2 are respectively 0.1 and 0.2 with probability 0.5 (case A), and 0.9 and 0.8 with probability 0.5 (case B). If you are not able to tell which case you face at any play, what is the best expectation of success you can achieve and how should you behave to achieve it? Now suppose that on each play you are told if you are facing case A or case B (although you still don't know the true action values). This is an associative search task. What is the best expectation of success you can achieve in this task, and how should you behave to achieve it?

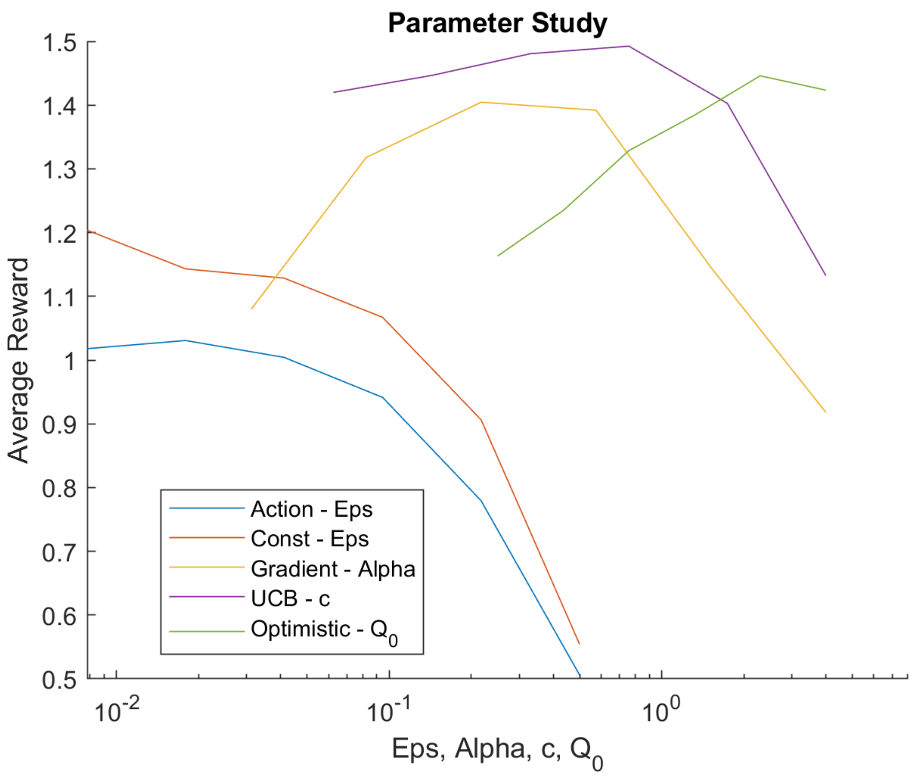
Clueless: The methods previously learned in the chapter would preform not well unless the true action values changed very slowly. However, since it is two levers, I would use either an optimistic greedy algorithm, or even an UCB because the levers produce a small state space.
Associative: In associative search tasking, we first look for some defining characteristic to tell us which player we are against. In the case of this scenario, we are told whether we are playing against case A or B. From there, we would use a policy that would tell us which actions are best for each case.

Exercise 2.11

Make a figure analogous to Figure 2.6 for the non-stationary case outlined in Exercise 2.5. Include the constant-step-size ε-greedy algorithm with $\alpha = 0.1$. Use runs of 200,000 steps and, as a performance measure for each algorithm and parameter setting, use the average reward over the last 100,000 steps.



Used increment algorithms for UCB and optimistic because they're faster than action value. Also did 6,000 steps because 200,000 takes forever. Note that the above graph was done for the first 1,000 steps, so we should expect the graphs to look somewhat different. (It was also for a stationary case, and the variances used were different).
Code after graph



Listing 2: Main Script – Parameter Study

```
% Initialize Variables
n = 10; % Number of arms
steps = 6000;
std = [0,sqrt(0.01)]; % [Mean, Sigma]
runs = 1000; % Number of runs to average over
points = 6; % How many points between points
start = (steps/2) + 1; % Where to start averaging the rewards

% Parameter Study Variables
epsilon = 2.^linspace(-7,-1,points);
alpha = 2.^linspace(-5,2,points);
c = 2.^linspace(-4,2,points);
Q_0 = 2.^linspace(-2,2,points);

% Initialize Rewards
final_rewards = zeros((steps/2),5);
rewards = zeros(points,5);

% Start Loop
for i = 1:points % Run through each point
    for k = 1:runs
        % Algoritms
        [r1,~] = Action_Value(steps,std,n,'moving','Average',0.1,epsilon(i↩
            ));
        [r2,~] = Action_Value(steps,std,n,'moving','Const',0.1,epsilon(i))↩
            ;
        [r3,~] = Gradient_Bandit(steps,std,n,'moving',alpha(i));
        [r4,~] = UCB(steps,std,n,'moving',c(i));
        [r5,~] = Optimistic(steps,std,n,'moving',Q_0(i),0); % Greedy ↩
            Optimstic eps = 0

        % Save half of the values
        final_rewards(:,1) = final_rewards(:,1) + r1(1,(start:steps))';
        final_rewards(:,2) = final_rewards(:,2) + r2(1,(start:steps))';
        final_rewards(:,3) = final_rewards(:,3) + r3(1,(start:steps))';
        final_rewards(:,4) = final_rewards(:,4) + r4(1,(start:steps))';
        final_rewards(:,5) = final_rewards(:,5) + r5(1,(start:steps))';
    end

    % Average the outputs
    final_rewards = final_rewards./runs;

    % Average the rewards
    rewards(i,1) = mean(final_rewards(:,1));
    rewards(i,2) = mean(final_rewards(:,2));
    rewards(i,3) = mean(final_rewards(:,3));
    rewards(i,4) = mean(final_rewards(:,4));
    rewards(i,5) = mean(final_rewards(:,5));

    % Empty final rewards for next iteration
    final_rewards = zeros((steps/2),5);
end

% Graphing
figure(3)
hold on
plot(epsilon,rewards(:,1)) % Average
plot(epsilon,rewards(:,2)) % Const
plot(alpha,rewards(:,3)) % Gradient
plot(c,rewards(:,4)) % UCB
plot(Q_0,rewards(:,5)) % Optimistic
legend('Action - Eps','Const - Eps','Gradient - Alpha','UCB - c','↩
   Optimistic - Q_0')
title('Parameter Study')
ylabel('Average Reward')
xlabel('Eps, Alpha, c, Q_0')
set(gca, 'XScale', 'log')
set(gca, 'YScale', 'linear')
xlim([0,8])
hold off
```