Problem 1.a

We know that:

> The call and exit floors are $(S_c), (S_e)$ $=\{1, 2, 3, 4, 5, 6\}$ and they can be on the same floor.
>
> The call and exit floors are $P(S_c)$ and $P(S_e|S_c)$ are the probability of someone calling that floor and the probability of someone exiting on a floor, respectively.

So we can say:

> a is the action taken, k is the number of times action a was selected, and W_{new}, W_{old} are the waittimes of the new and old elevator.
>
> $r((S_c), (S_e))$ is a reward function for the simulation, and $r((a), (S_e))$ is the reward function for the action taken.

Then we can define a **utility** function:

> Utility = $Q_k(a) = Q_k(a) + \frac{1}{k+1}(r(S_c, S_e)_{k+1} - Q_k(a))$

and the **reward** function as:

> $r(S_c, S_e) = min(W_{new}, W_{old})$
>
> $= max(-W_{new}, -W_{old}) =$
>
> $= max(-(5|(S_e) - (S_c)| + 2(7)), -(7|(S_e) - (S_c)| + 2(7)))$

Problem 1.b

Given that we need to determine the ideal floor, and we do not know the distribution of people who start on each floor, we can define the $arms$ of the bandit as the floors:

> **arms of bandit =** $\{P_s(1), P_s(2), P_s(3), P_s(4), P_s(5), P_s(6)\}$

and to determine the average reward for each arm, we use the utility function:

> Utility = $Q_k(a) = Q_k(a) + \frac{1}{k+1}(r((S_c),(S_e))_{k+1} - Q_k(a))$

In which this **utility** fucntion uses the reward function:

> $r((S_c),(S_e)) = min(W_{new}, W_{old}) = max(-W_{new}, -W_{old}) =$

Where this reward function chooses the max negative waitime between the old and new eleavtor, ensuring the fastest elevator comes. And since there is enough time between calls to ensure neither elevator will be used at the same time, this reward function holds.

After recieving the highest utility, we need to choose the corresponding action:

> $a_{greedy} = argmax_a(Q_k(a))$

Lastly, we need to find a policy for the agent to follow. We can use the ϵ-greedy solution.

We will have different ϵ values and run those simulations for N Steps/iterations

> $\epsilon = .1, .2, .3 ... 1$, Where ϵ = .1 means we choose other floors 10% of the time, and ϵ = 1, means we choose each floor equally. And we choose the best floor so far 1-ϵ of the time

We these definitions, we have a full n-armed bandit probelm that can be solved.

Problem 1.c.i

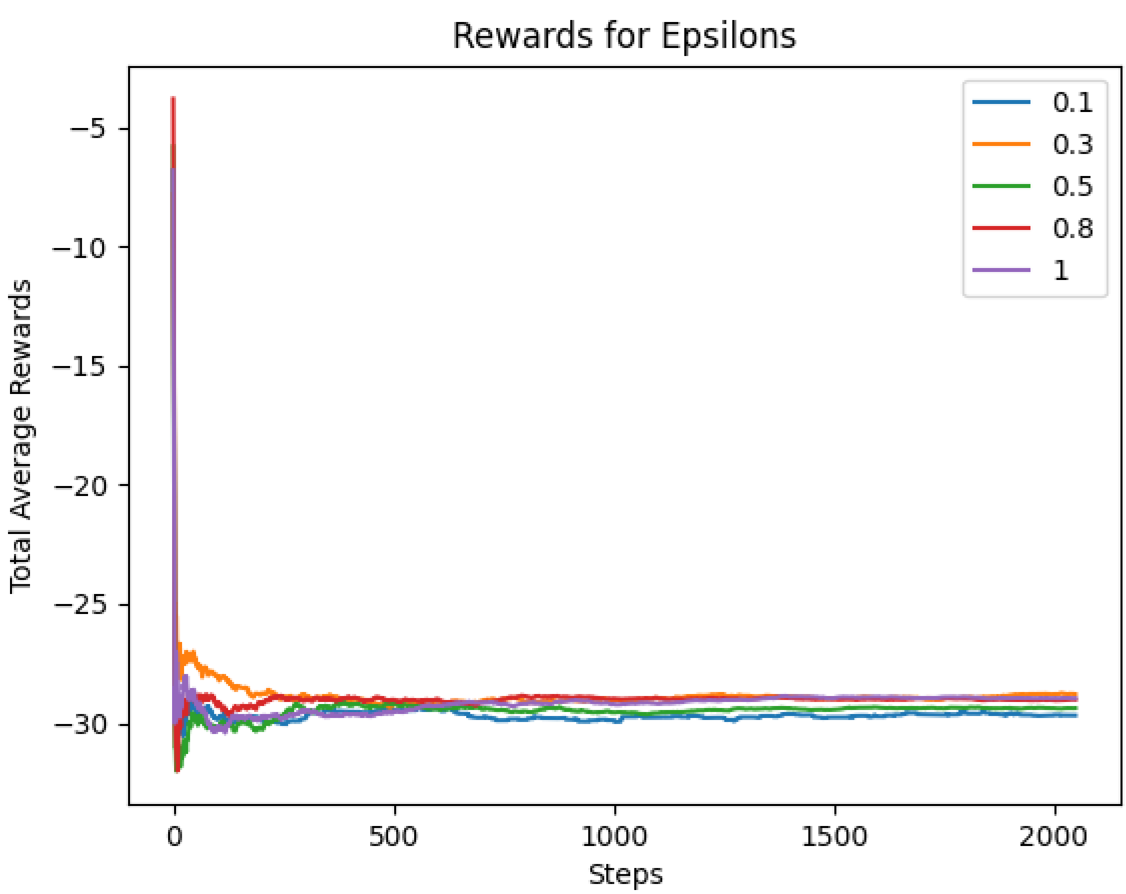
In this code, all people come in on the first floor, and exit on floors (2-6) uniformly. The piece of code in interest is

```
START_FLOORS = [1] Start on floor one
START_PROB = [1] 100% start on floor one
EXIT_FLOORS = [2,3,4,5,6] Exit on floors 2-6
EXIT_PROB = [.20, .20, .20, .20, .20] Uniform exit
```

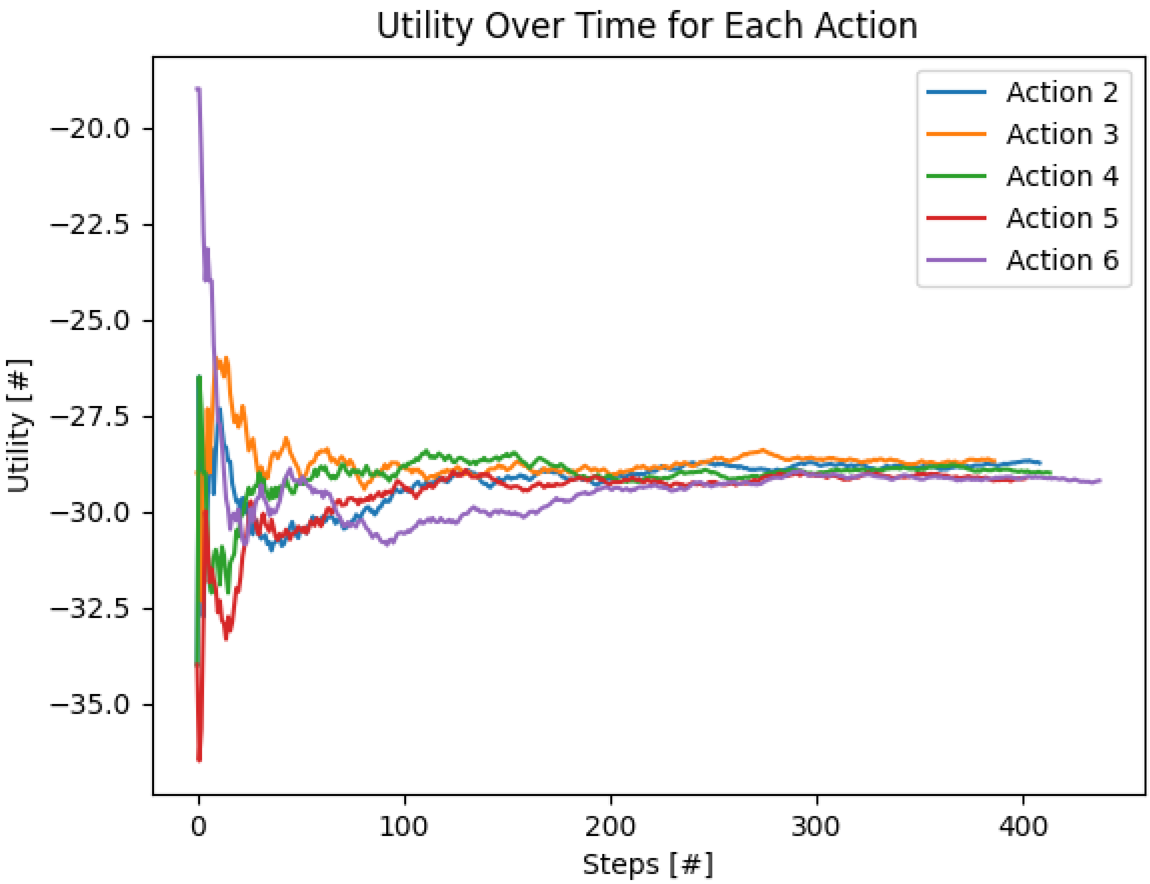
Given those paramaters, here were the results:

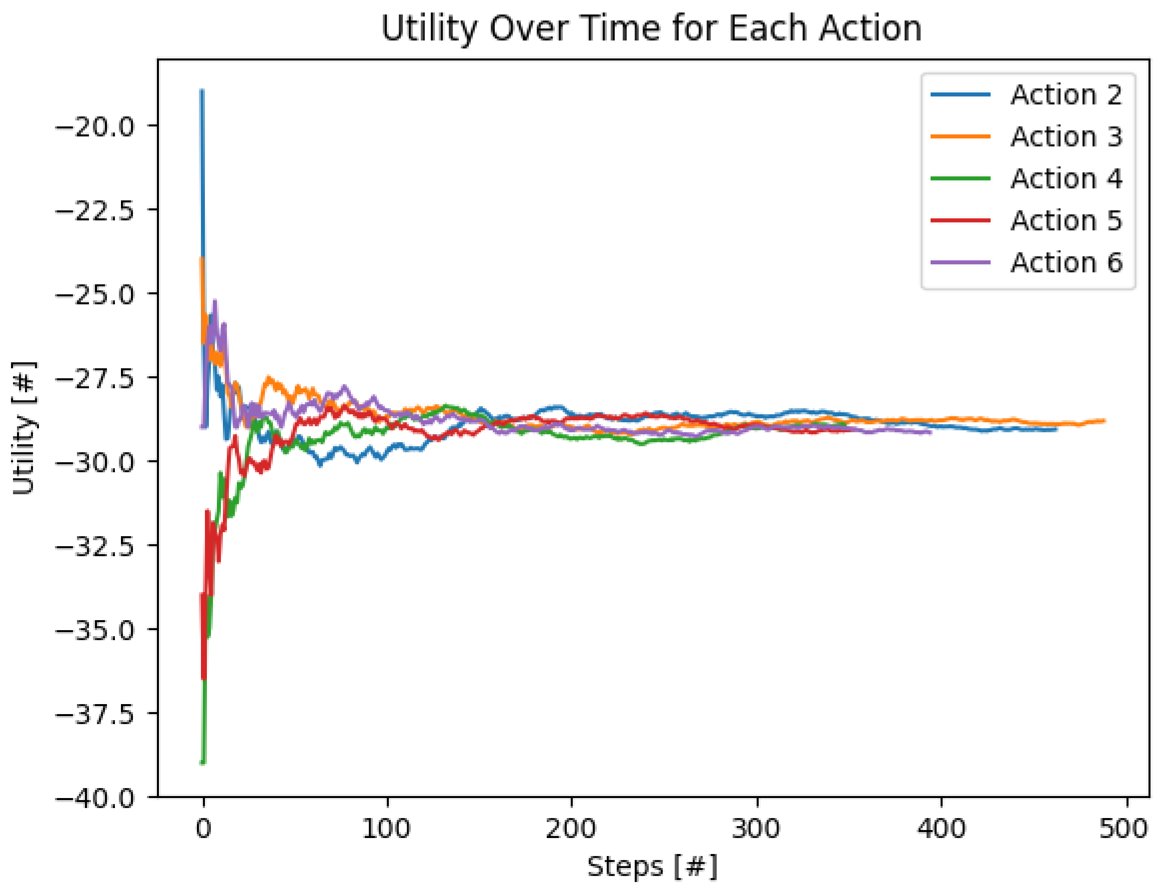
```
-----------------------------------
Epsilon (0.1) with a total average reward of -29.5902. Best start floor = 4
Epsilon (0.3) with a total average reward of -29.3658. Best start floor = 5
Epsilon (0.5) with a total average reward of -29.3654. Best start floor = 5
Epsilon (0.8) with a total average reward of -28.5935. Best start floor = 2
Epsilon (1) with a total average reward of -28.8558. Best start floor = 6
-----------------------------------
Best floor was 2 with avg utility of -28.5935 over 2000 steps and 5 experiments.
```

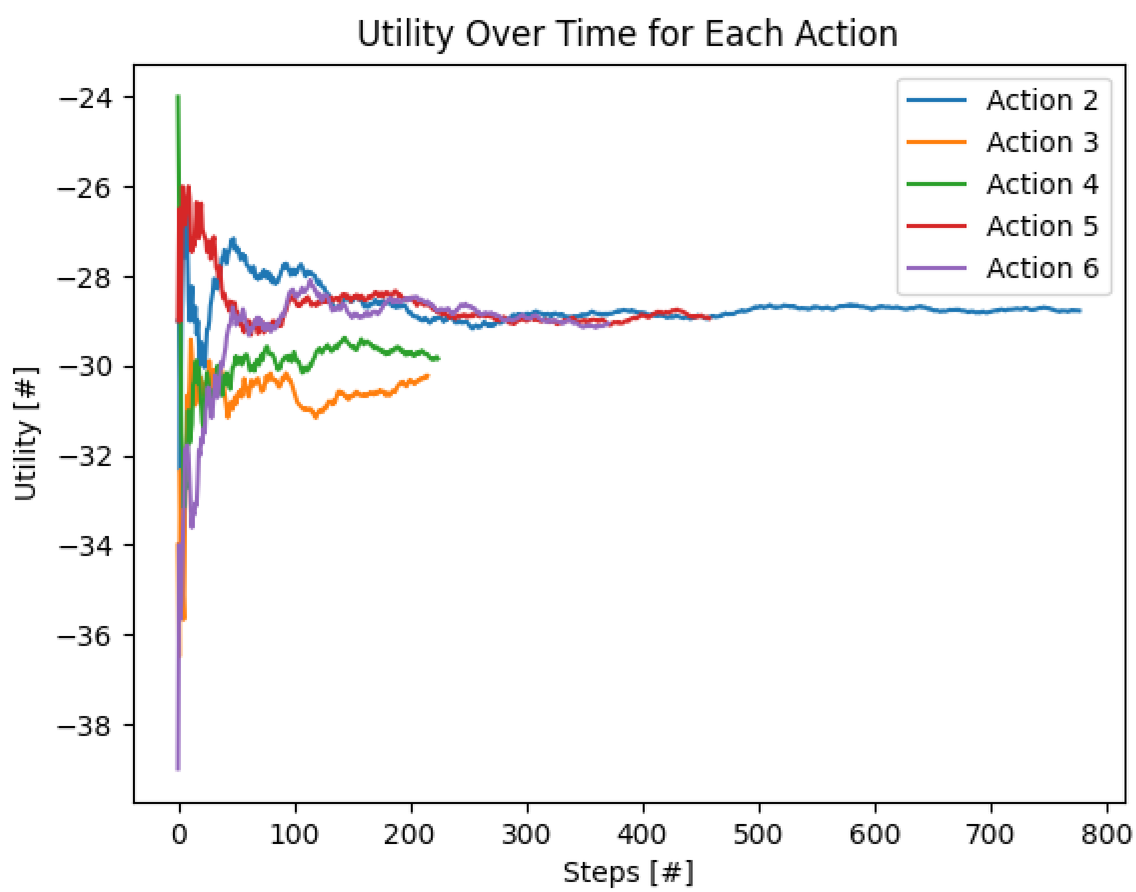
1.c.i Graph: Epsilons

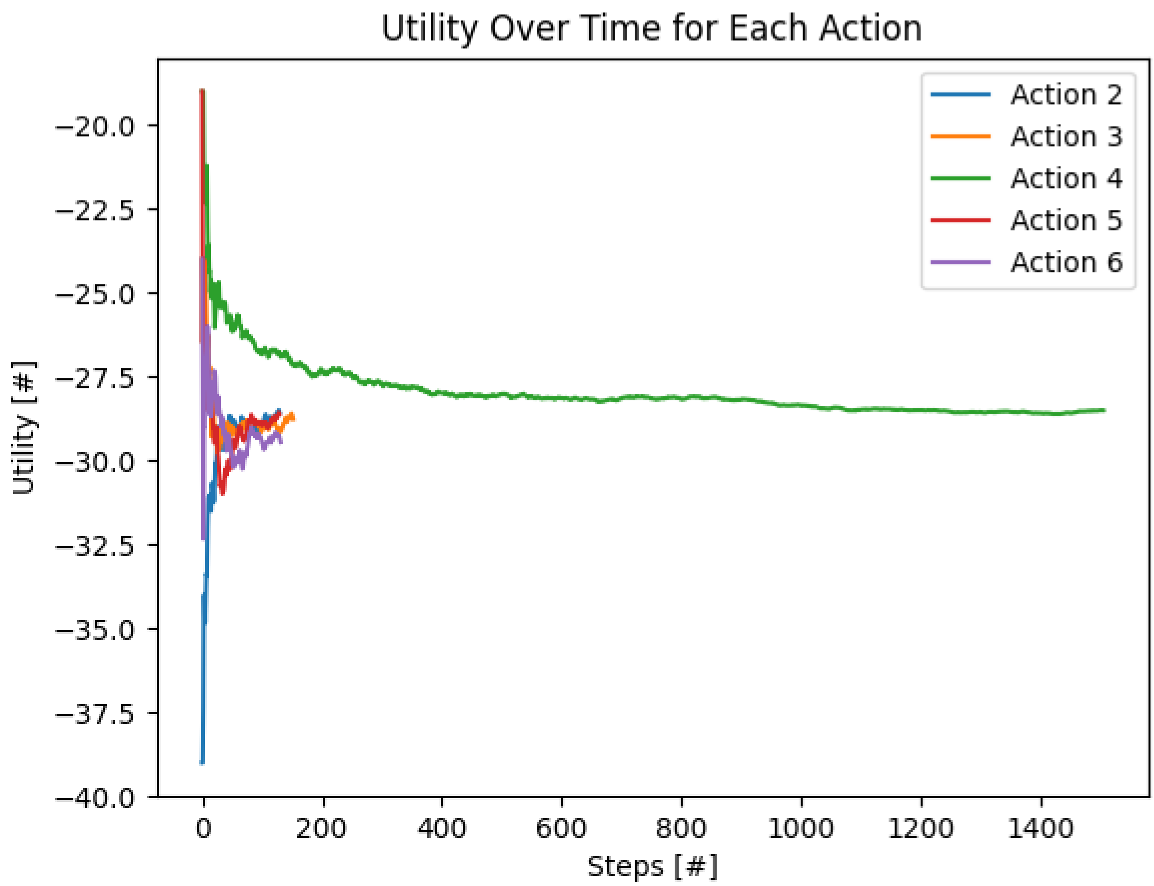


1.c.i Graph: Utilities ϵ = (1.0, 0.80, 0.50, 0.30, 0.10)

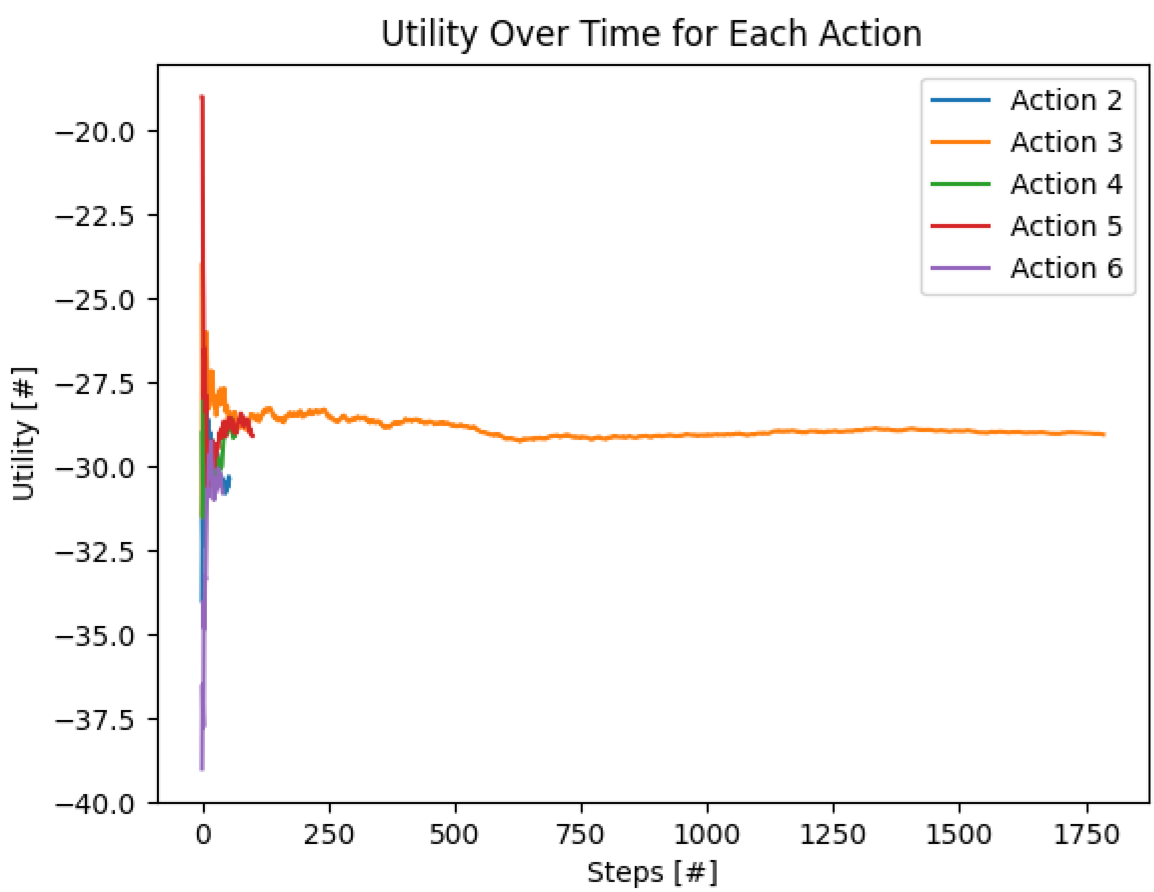



Utility Over Time for Each Action
Action 2
Action 3
Action 4
Action 5
Action 6
−20.0
−22.5
−25.0
−27.5
−30.0
−32.5
−35.0
−37.5
−40.0
Utility [#]
0
100
200
300
400
500
Steps [#]


Utility Over Time for Each Action
Action 2
Action 3
Action 4
Action 5
Action 6
−24
−26
−28
−30
−32
−34
−36
−38
Utility [#]
0
100
200
300
400
500
600
700
800
Steps [#]



Utility Over Time for Each Action
Action 2
Action 3
Action 4
Action 5
Action 6
−20.0
−22.5
−25.0
−27.5
−30.0
−32.5
−35.0
−37.5
−40.0
Utility [#]
0
200
400
600
800
1000
1200
1400
Steps [#]



Utility Over Time for Each Action
Action 2
Action 3
Action 4
Action 5
Action 6
−20.0
−22.5
−25.0
−27.5
−30.0
−32.5
−35.0
−37.5
−40.0
Utility [#]
0
250
500
750
1000
1250
1500
1750
Steps [#]

Basic Code Structure for Problem 1.c-e and 2.a-b

Below is the basic code structure that will be used trhoughout with section modified for each problem

In [ ]:

```python
import os
import numpy as np
import matplotlib.pyplot as plt

def graph_loss(actions, losses, number, title):
    for action in actions:
        plt.plot([i for i in range(len(losses[action]))], losses[action], label
=f'Action {action}')

    plt.xlabel('Steps')
    plt.ylabel('loss')
    plt.title('loss Over Time for Each Action')
    plt.legend()
    plt.savefig(title + '_loss_' + str(number) + '_.png')
    plt.show()

def graph_actions(actions, util_history, number, title):

    for action in actions:
        plt.plot([i for i in range(len(util_history[action]))], util_history[ac
tion], label=f'Action {action}')

    plt.xlabel('Steps [#]')
    plt.ylabel('Utility [#]')
    plt.title('Utility Over Time for Each Action')
    plt.legend()
    plt.savefig(title + '_utility_' + str(number) + '_.png')
    plt.show()


def graph_epsilons(experiemnts, epsilons, title):
    for k in range(len(experiemnts)):
        plt.plot([i for i in range(len(experiemnts[k]))], experiemnts[k], label
=str(epsilons[k]))

    plt.xlabel('Steps')
    plt.ylabel('Total Average Rewards')
    plt.title('Rewards for Epsilons')
    plt.legend()
    plt.savefig(title + '_epsilons.png')
    plt.show()

START_FLOORS = [1]
EXIT_FLOORS = [2,3,4,5,6]
START_PROB = [1]
EXIT_PROB = [.20, .20, .20, .20, .20]
VERBOSE = True
ACTIONS = [2, 3, 4, 5, 6]  # action choices
EXPERIMENTS = 10
STEPS = 10
EPSILONS = [.1, .5, 1]
TITLE = "problem_1_c_i"


epsilon_experiment_values = []

class ElevatorSimulation:

    def __init__(self, explore_probability):

        self.explore_probability = explore_probability
```

```
        self.exploit_probability = 1 - explore_probability
        self.q_history_each = {i: [] for i in range(1, 6 + 1)}
        self.action_sum_history = []
        self.call_floors_count = {s: 0 for s in range(6 + 1)}
        self.exit_floors_count = {s: 0 for s in range(6 + 1)}
        self.loss = {i: [] for i in range(1, 6 + 1)}
        self.count = {a: 0 for a in ACTIONS}
        self.q_val = {r: 0 for r in ACTIONS}
        self.avg_q_values = [] # List of avg q values over all actions for each
step

    def q_func(self, action, actual, r_hat):
        """
        Utility function (Expected long term reward)
        Q(a)_k+1 = Q(a)_k + (1/k+1) * (r_k+1 - Q(a)_k )

        Args:
            action (int): floor chosen
            r_k (int): actual reward from simulation
            r_hat (int): predicted reward (not used)
        """

        self.q_val[action] = self.q_val[action] + (1 / (self.count[action] + 1)
) * (actual - self.q_val[action])
        self.loss[action].append(self.q_val[action] + (1 / (self.count[action]
+ 1) ) * (actual - r_hat))
        self.count[action] += 1

        self.q_history_each[action].append(self.q_val[action])
        self.avg_q_values.append(sum(self.q_val.values())/len(ACTIONS))

    def reward_func(self, s_c, s_e):
        """
        Calcualtes reward based on minimum time it takes between old and new el
evator.
        Args:
            s_c (int): call elevator
            s_e (int): exit elevator

        Returns:
            reward: min time between call and exit floor (-)
        """

        time_new = 5 * abs(s_c - s_e) + (2 * 7)
        time_old = 7 * abs(s_c - s_e) + (2 * 7)
        max_reward = max(-time_new, -time_old)

        return int(max_reward)

    def epsilon_greedy(self):
        """
        Epsilon greedy policy.
        Choose random floor with P() = epsilon \n
        Choose action that provided max util with P() = 1 - epsilon \n
        Returns: int: action chosen
        """

        policy = np.random.choice(['explore', 'exploit'], 1, p=[self.explore_pr
obability, self.exploit_probability])

        if policy == 'explore':
            return np.random.choice(ACTIONS)
```

```
        else:
            return max(self.q_val, key=self.q_val.get)

    def simulate(self):
        """
        Simulates people chosing an elevator.
        Gets a random call and exit floor from the list call and exit floors\n
        Returns: (int, int): call floor and exit floor the person chose in simu
altion
        """

        call_floor = np.random.choice(START_FLOORS, 1, p=START_PROB)
        exit_floor = np.random.choice(EXIT_FLOORS, 1, p=EXIT_PROB)

        self.exit_floors_count[int(exit_floor)] += 1
        self.call_floors_count[int(call_floor)] += 1

        return call_floor, exit_floor

    def print_agent(self, action, e_floor, c_floor, r, r_hat):
        """Prints detaisl about agent

        Args:
            action (int): action chosen
            e_floor (int): exit floor
            c_floor (int): call floor
            r (int): reward actual
            r_hat (int): reward prediction
        """

        print("------------------------------------")
        print(f"Actions Count = {self.count}")
        print(f"Q_array = {self.q_val}")
        print(f"Floors: Sc = {c_floor}, Se = {e_floor}")
        print(f"Actual Reward = r(Sc={c_floor}, Se={e_floor}) = {r}")
        print(f"Predicted Reward = r(Sc={action}, Se={e_floor})= {r_hat}")
        print(f"Q({action}) = {self.q_val[action]}")
        print("------------------------------------")
        print(f"argmax Q_array = {max(self.q_val, key=self.q_val.get)}")
        print("------------------------------------")

    def run(self):
        """Runs an agent with given epsilon value for the given amount of step
s"""

        # Gets initial values for each action
        for _ in range(EXPERIMENTS):
            for a in ACTIONS:
                start_floor, exit_floor = self.simulate()
                actual_reward = self.reward_func(start_floor, exit_floor)
                predicted_reward = self.reward_func(a, exit_floor)
                self.q_func(a, actual_reward, predicted_reward)

        # Runs for the number of steps and gets q values
        for _ in range(STEPS):
            action = self.epsilon_greedy()
            start_floor, exit_floor = self.simulate()
            actual_reward = self.reward_func(start_floor, exit_floor)
            predicted_reward = self.reward_func(action, exit_floor)
            self.q_func(action, actual_reward, predicted_reward)
            self.print_agent(action, exit_floor, start_floor, actual_reward, pr
edicted_reward)
```

```
        print(f"Action floor count = {self.count}")
        print(f"Exit floor count = {self.exit_floors_count}")
        print(f"Call floor count = {self.call_floors_count}")

        epsilon_experiment_values.append(self.avg_q_values)

        if VERBOSE:
            graph_actions(ACTIONS, self.q_history_each, len(epsilon_experiment_
values), TITLE)
            graph_loss(ACTIONS, self.loss, len(epsilon_experiment_values), TITL
E)

if __name__ == "__main__":

    agents = []
    best_floors = {}

    for e in range(len(EPSILONS)):
        agents.append(ElevatorSimulation(EPSILONS[e]))
        agents[e].run()

    graph_epsilons(epsilon_experiment_values, EPSILONS, TITLE)

    print("------------------------------------")
    for k in range(len(epsilon_experiment_values)):
        print(f"Epsilon ({agents[k].explore_probability}) with a total average
reward of {round(np.mean(epsilon_experiment_values[k]),4)}. Best start floor =
{max(agents[k].q_val, key=agents[k].q_val.get)}")
        best_floors[round(np.mean(epsilon_experiment_values[k]),4)] = max(agent
s[k].q_val, key=agents[k].q_val.get)
    print("------------------------------------")
    print(f"Best floor was {best_floors[max(best_floors)]} with avg utility of
{max(best_floors)} over {STEPS} steps and {len(epsilon_experiment_values)} expe
riments.")
```

Problem 1.c.ii

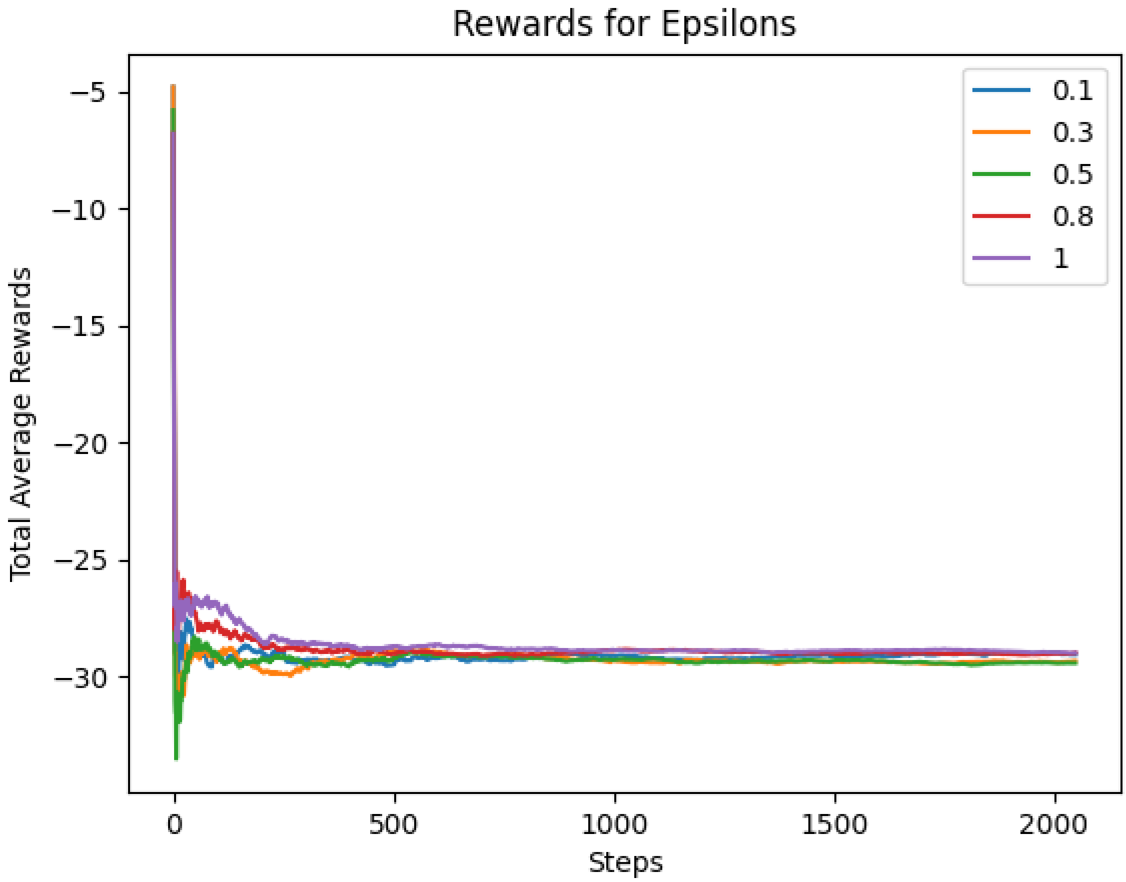
This problem is similar to the first since people call the elevator from floors 2-6 with a uniform distribution and all exit on the first floor. So here is the only change made:

```
START_FLOORS = [2,3,4,5,6] # call from 2- 6
START_PROB = [.20, .20, .20, .20, .20] # uniform dist
EXIT_FLOORS = [1] # exit on floor 1
EXIT_PROB = [1] # 100% chance exit on floor 1
VERBOSE = True
ACTIONS = [2, 3, 4, 5, 6]  # action choices
EXPERIMENTS = 10
STEPS = 2000
EPSILONS = [.1, .3, .5, .8, 1]
TITLE = "problem_1_c_i"
```

Here were the results with the params run:

```
-----------------------------------
Epsilon (0.1) with a total average reward of -29.0986. Best start floor = 3
Epsilon (0.3) with a total average reward of -29.2614. Best start floor = 3
Epsilon (0.5) with a total average reward of -29.2761. Best start floor = 6
Epsilon (0.8) with a total average reward of -28.817. Best start floor = 5
Epsilon (1) with a total average reward of -28.6456. Best start floor = 2
-----------------------------------
Best floor was 2 with avg utility of -28.6456 over 2000 steps and 5 experiments.
```

1.c.ii Graph: Epsilons


Rewards for Epsilons
Total Average Rewards
Steps
−5
−10
−15
−20
−25
−30
0
500
1000
1500
2000
0.1
0.3
0.5
0.8
1

Problem 1.c.iii

This probelm is slightly more complicated. This one gets more involved with conditionaly proabability. I split the groups of people into two:

Group one

> 50% of the time the elevator is called from the second floor where people want to go to the other floors with a uniform distribution while the other

Group two

> 50% of the time the elevator is called from floors 2-6 with a uniform distribution with persons always exiting on the 1st floor

How I accomplished this is by choosing one of the groups:

```
worker = np.random.choice(['call-from-floor-2', 'call-from-floor-2-to-6'], 1, p=[.50, .50])
```

Then depending on which worker group was called, I chose that respective distribution:

```
if worker == 'call-from-floor-2':
    START_FLOORS = [2]
    EXIT_FLOORS = [1,3,4,5,6]
    START_PROB = [1]
    EXIT_PROB = [.20, .20, .20, .20, .2]

    call_floor = np.random.choice(START_FLOORS, 1, p=START_PROB)
    exit_floor = np.random.choice(EXIT_FLOORS, 1, p=EXIT_PROB)

else:
    START_FLOORS = [2,3,4,5,6]
    EXIT_FLOORS = [1]
    START_PROB = [.20, .20, .20, .20, .20]
    EXIT_PROB = [1]
    call_floor = np.random.choice(START_FLOORS, 1, p=START_PROB)
    exit_floor = np.random.choice(EXIT_FLOORS, 1, p=EXIT_PROB)
```

Here were the parmaters and results run with the experiment:

```
VERBOSE = False
ACTIONS = [2, 3, 4, 5, 6]  # action choices
EXPERIMENTS = 10
STEPS = 2000
EPSILONS = [.1, .1, .3, .3, .5, .5, .8, .8, 1, 1]

TITLE = "_1_c_iii"
```

```
-----------------------------------
Epsilon (0.1) with a total average reward of -27.7886. Best start floor = 3
Epsilon (0.1) with a total average reward of -27.7101. Best start floor = 5
Epsilon (0.3) with a total average reward of -26.9955. Best start floor = 6
Epsilon (0.3) with a total average reward of -27.0921. Best start floor = 2
Epsilon (0.5) with a total average reward of -26.6924. Best start floor = 4
Epsilon (0.5) with a total average reward of -27.5851. Best start floor = 6
Epsilon (0.8) with a total average reward of -27.381. Best start floor = 3
Epsilon (0.8) with a total average reward of -27.3792. Best start floor = 4
Epsilon (1) with a total average reward of -27.0863. Best start floor = 6
Epsilon (1) with a total average reward of -26.7973. Best start floor = 4
-----------------------------------
Best floor was 4 with avg utility of -26.6924 over 2000 steps and 10 experiments.
```

Problem 1.c.i - iii Explanations/Conclusions

The new elevator was exclusively used, because of the reward function used:

> $$r((S_c),(S_e)) = min(W_{new}, W_{old}) = max(-W_{new}, -W_{old}) =$$
> $$= max(-(5|(S_e)-(S_c)|+2(7)), -(7|(S_e)-(S_c)|+2(7)))$$

This ensures that the elevator which minimizes the waittime will be the fastest elevator, which is the new elevator. This happened for each experiement and for each problem.

Problem 1.d

The problem description for this problem has a few logical and grammatical ambiguities but I did the best I could to understand the problem.

This solution assumes

> The $P(S_c)$ or calling an elevator is uniform as stated in the problem description.
>
> The problablity of going down $P_{down}(S_e)$ increases $2x$ for each floor it crosses (linear increase in time).
>
> When you go up, the for each floor going up: $P_{up}(S_e) = max(P_{down}(S_e))$. Or in other words. The probability of going up is the same as going all the way down, as stated in the assignment.
>
> A person will **NOT** exit and start on the same floor, logically.

Here is the mathematical solution

> $e = .10$
>
> $S_{call} = P(S_c)$
>
> $P_{upper}(S_{call}) = [e_{S_{call}}, 0, \ldots, 0]$
>
> $P_{lower}(S_{call}) = [0, 0, \ldots, e_{S_{call}}]$
>
> $P_{lower}(S_{call}) = [2 * e_{S_{call}-1}, \ldots, 2 * e_{S_{call}-1}, e_{S_{call}}]$
>
> $P_{upper}(S_{call}) = [e_{S_{call}}, max(P_{lower}), \ldots, max(P_{lower})]$
>
> $P(S_e) = P_{lower}(S_{call}) \cup P_{upper}(S_{call})$
>
> $P(S_e) = Normalize(P(S_e))$; where $\Sigma P(S_e)_i = 1$

And for clarity. here is an example output for deciding the exit floor:

```
Call Floor: 4 # starting floor
e: 0.1 # scalar value to start with
upper_probs: [0.1, 0, 0] # going up
lower_probs: [0, 0, 0, 0.1] # going down

unormalized: [0.8, 0.4, 0.2, 0, 0.8, 0.8]
normalized: [0.26, 0.13, 0.066, 0.0, 0.26, 0.26]

exit_floor: [5]
```

Here is the code that was modified with part d:

```python
    def simulate(self):
        """
        Simulates people chosing an elevator.
        Gets a random call and exit floor from the list call and exit floors\n
        Returns: (int, int): call floor and exit floor the person chose in simualtion
        """

        call_floor = np.random.choice(START_FLOORS, 1, p=START_PROB)

        e = 0.1 # scalr value
        call_floor = int(np.random.choice(START_FLOORS, 1, p=START_PROB))

        e_probs = [0, 0, 0, 0, 0, 0]
        e_probs[call_floor-1] = e

        upper_probs = e_probs[call_floor-1:]
        lower_probs = e_probs[:call_floor]

        for i in range(len(lower_probs)-1, -1, -1):
            if i - 1 < 0:
                break
            else:
                lower_probs[i - 1] = 2 * lower_probs[i]

        for i in range(len(upper_probs)):
            if len(lower_probs) > 0:
                upper_probs[i] = lower_probs[0]

        lower_probs.pop()

        unnormalized = lower_probs + upper_probs
        unnormalized[call_floor-1] = 0

        # Normalize the probabilities
        normalized_probs = [p / sum(unnormalized) for p in unnormalized]

        exit_floor = np.random.choice(EXIT_FLOORS, 1, p=normalized_probs)

        self.exit_floors_count[int(exit_floor)] += 1
        self.call_floors_count[int(call_floor)] += 1

        return call_floor, exit_floor
```

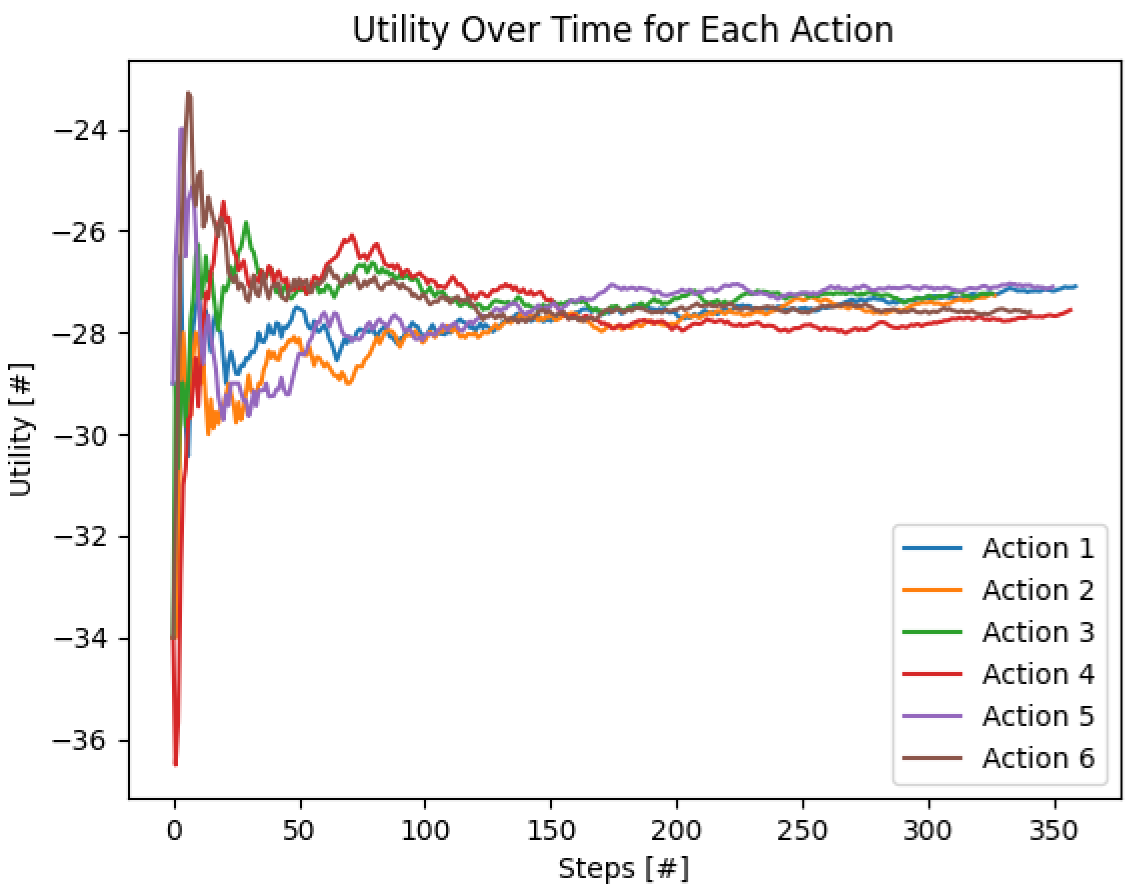
Here are the results:

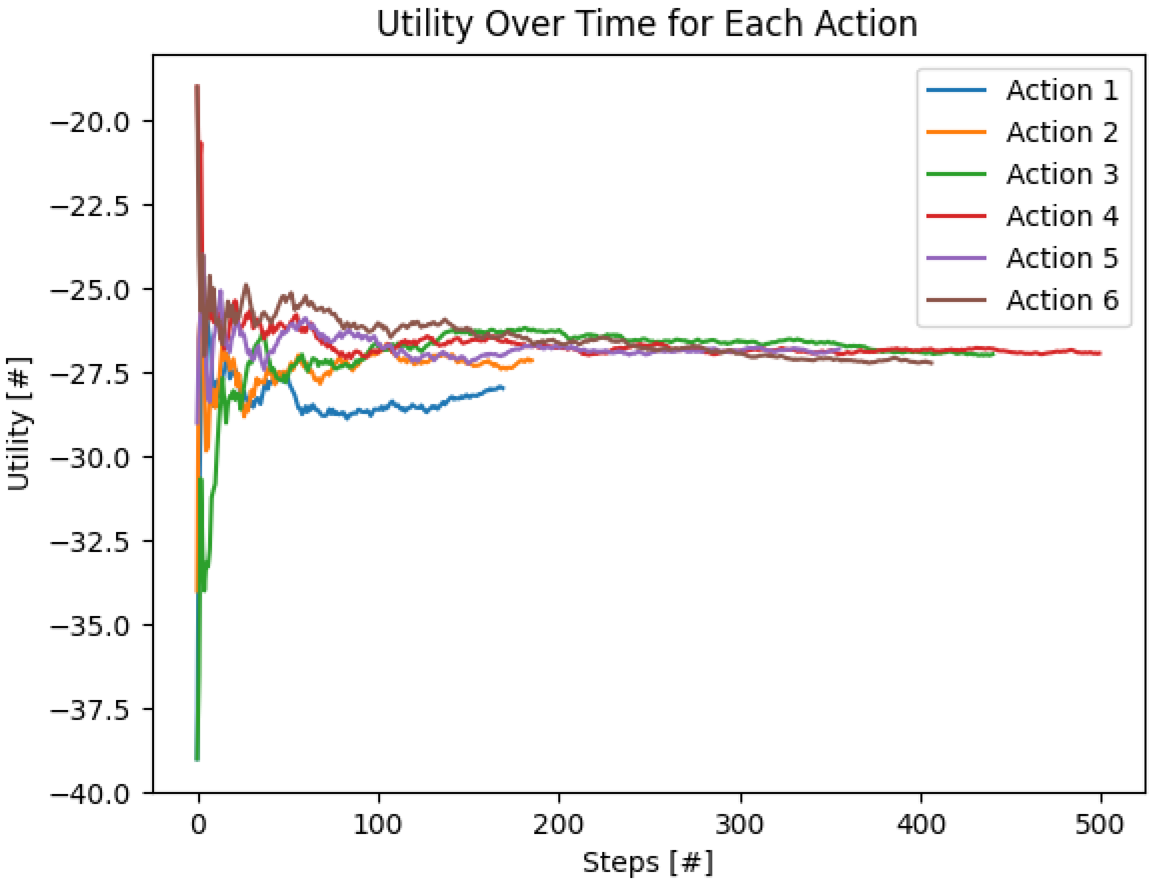
```
-----------------------------------
Epsilon (0.1) with a total average reward of -27.5404. Best start floor = 3
Epsilon (0.5) with a total average reward of -27.1953. Best start floor = 5
Epsilon (1) with a total average reward of -27.5429. Best start floor = 1
-----------------------------------
Best floor was 5 with avg utility of -27.1953 over 2000 steps and 3 experiments.
```

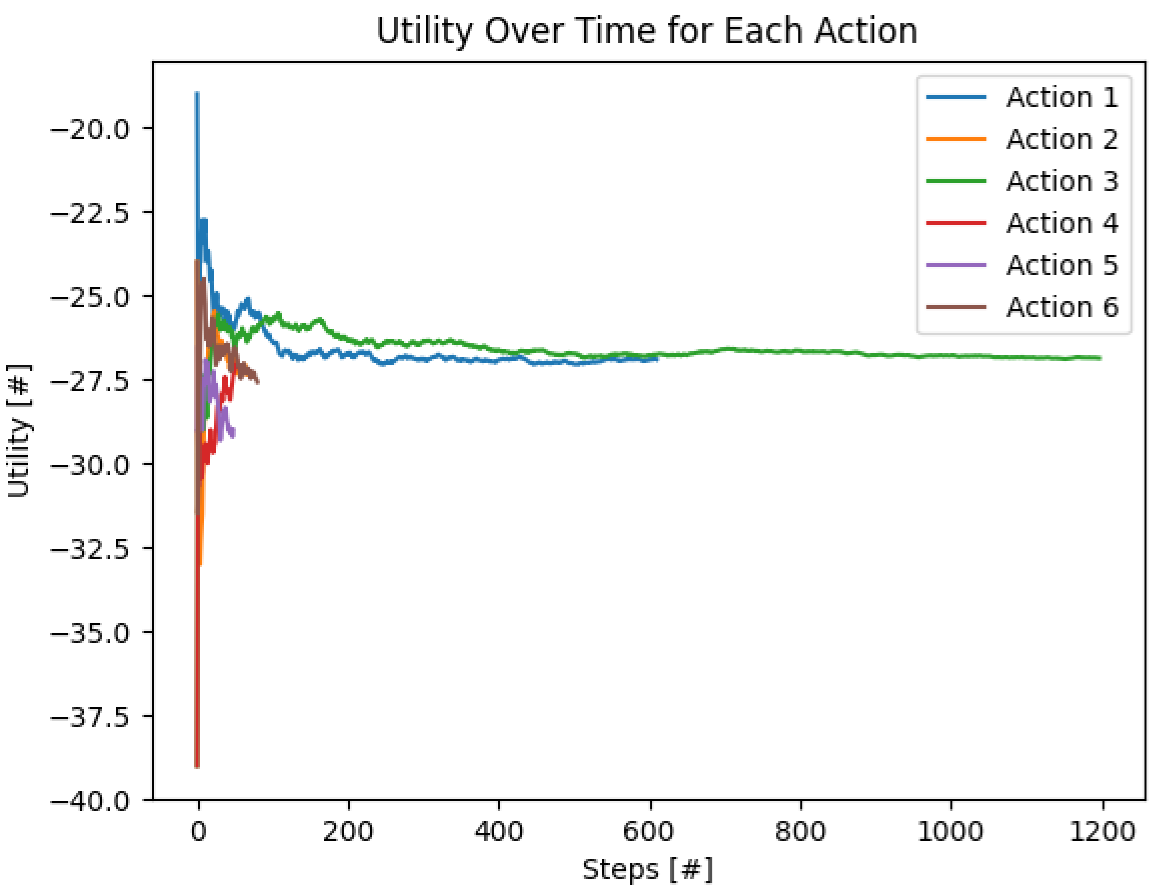
Here are the graph results as well

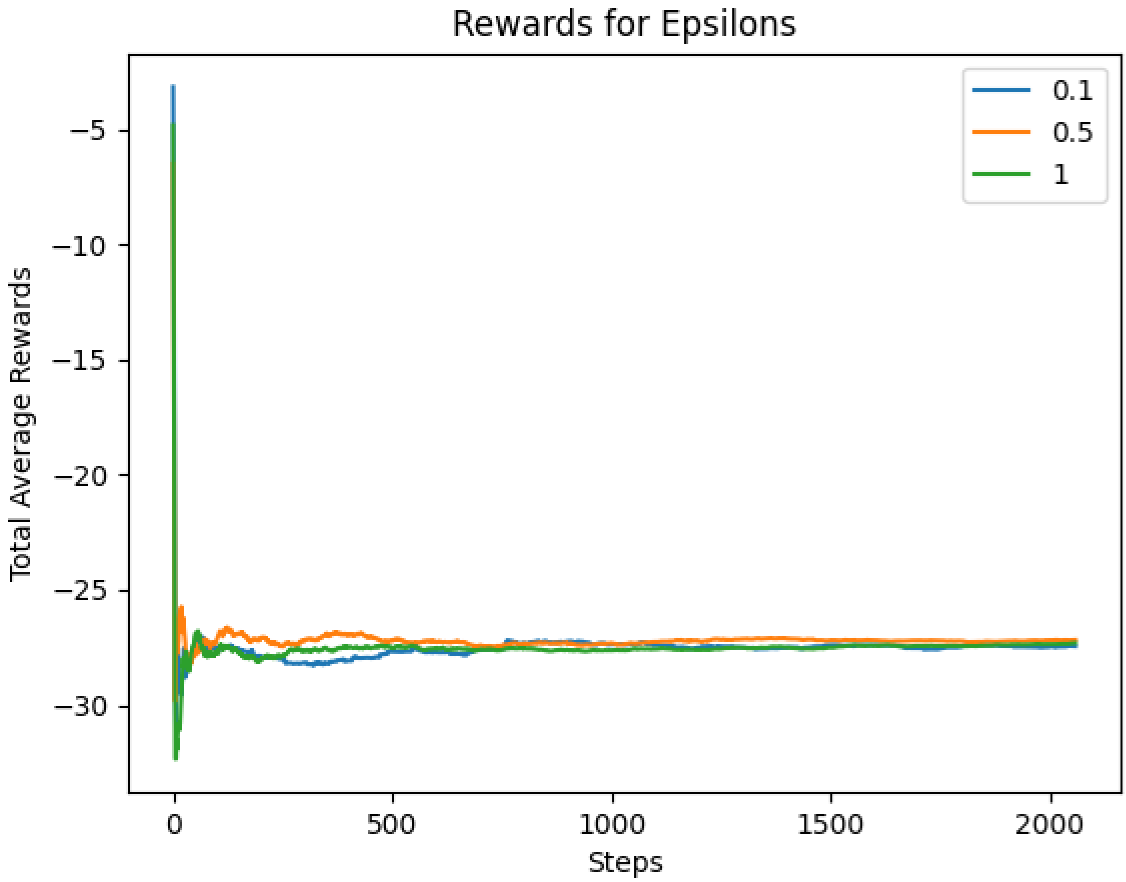
1.d Graph: Utilities ϵ = (1.0, 0.50, 0.10)

1.d Graph: Epsilon




Utility Over Time for Each Action
Action 1
Action 2
Action 3
Action 4
Action 5
Action 6
Utility [#]
Steps [#]
−20.0
−22.5
−25.0
−27.5
−30.0
−32.5
−35.0
−37.5
−40.0
0
100
200
300
400
500


Utility Over Time for Each Action
Action 1
Action 2
Action 3
Action 4
Action 5
Action 6
Utility [#]
Steps [#]
−20.0
−22.5
−25.0
−27.5
−30.0
−32.5
−35.0
−37.5
−40.0
0
200
400
600
800
1000
1200



Problem 1.d Explanations/Conclusions

The final eleavtor chosen was exclusively the new eleavtor of course, and it believed floor 5 was the best elevator. Better results can of course be obtained by increasing the steps and increasing the number and range of epsilon values.

Problem 1.e

This problem is very similar to 1.c.iii in which it emphasizes conditionaly probability. Since So I had to split the people into groups first.

Day workers

> 90% of all people arrive arrive on the first floor and travel to each of the other floors (2-6) with uniform probability;

Night workers

> he remaining 10% (late night workers) push the button on an upper floor (2-6) with uniform probability and push the button for floor 1

Here is the modified piece of code:

```
 worker = np.random.choice(['night', 'day'], 1, p=[.10, .90])

if worker == 'night':
    START_FLOORS = [2,3,4,5,6]
    EXIT_FLOORS = [1]
    START_PROB = [.20, .20, .20, .20, .20]
    EXIT_PROB = [1]


    call_floor = np.random.choice(START_FLOORS, 1, p=START_PROB)
    exit_floor = np.random.choice(EXIT_FLOORS, 1, p=EXIT_PROB)

# day workers
else:
    START_FLOORS = [1]
    EXIT_FLOORS = [2, 3, 4, 5, 6 ]
    START_PROB = [1]
    EXIT_PROB = [.20, .20, .20, .20, .20]
    call_floor = np.random.choice(START_FLOORS, 1, p=START_PROB)
    exit_floor = np.random.choice(EXIT_FLOORS, 1, p=EXIT_PROB)
```

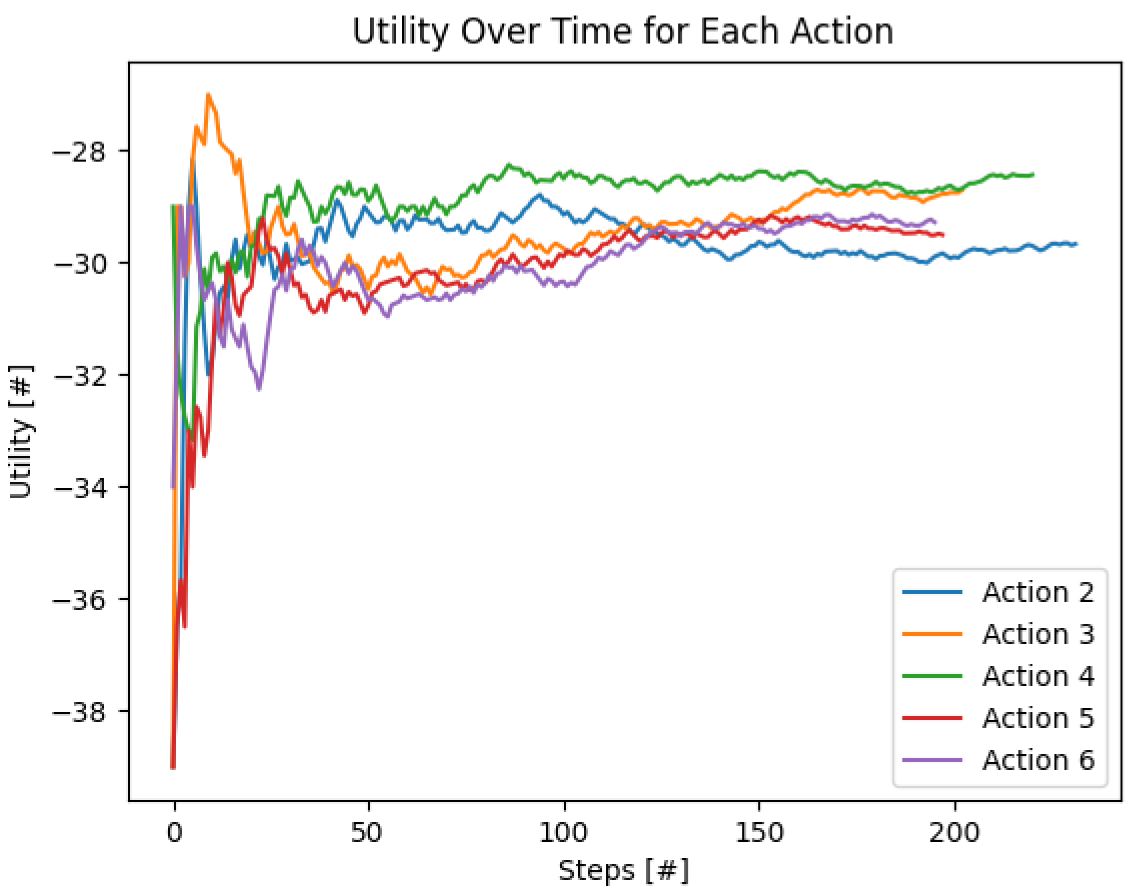
Here are the results:

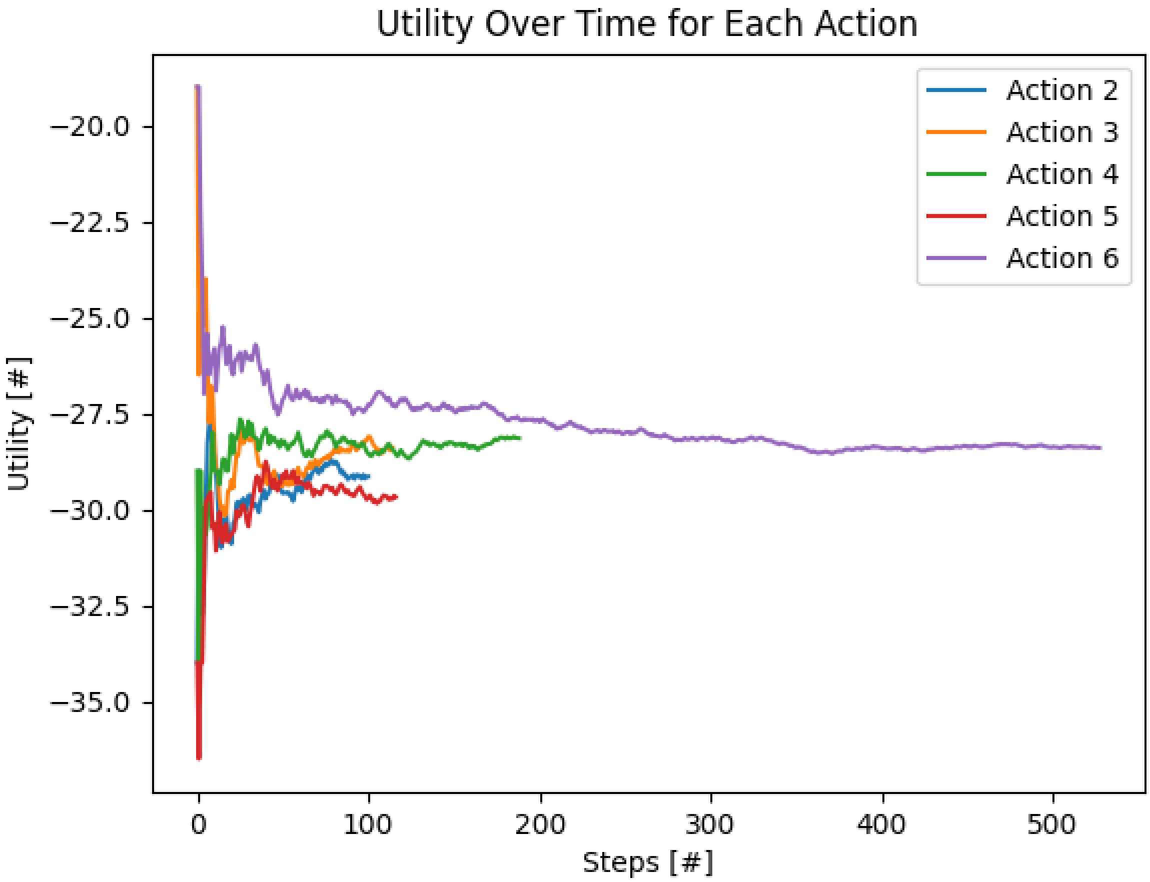
```
VERBOSE = True
ACTIONS = [2, 3, 4, 5, 6]  # action choices
EXPERIMENTS = 10
STEPS = 1000
EPSILONS = [.1, .5, 1]
```

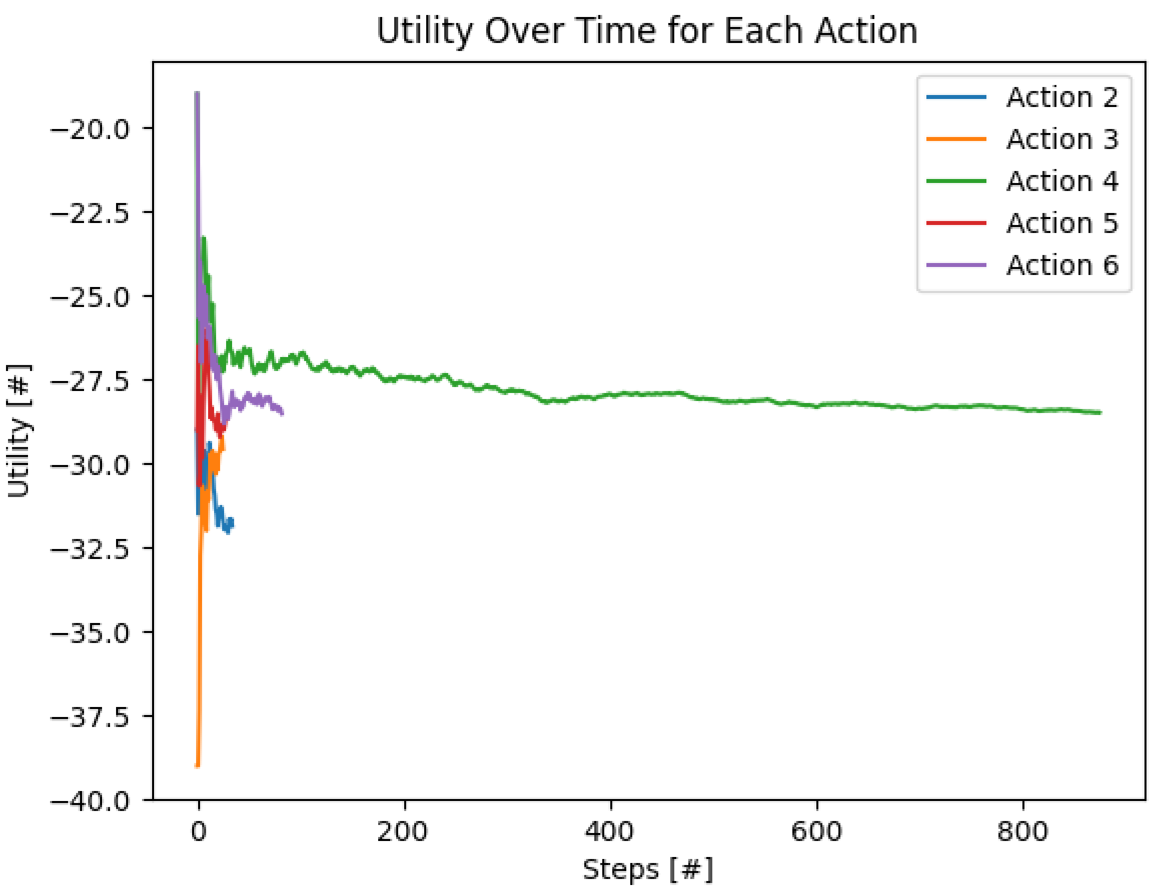
```
-----------------------------------
Epsilon (0.1) with a total average reward of -29.1551. Best start floor = 4
Epsilon (0.5) with a total average reward of -28.7686. Best start floor = 4
Epsilon (1) with a total average reward of -29.5691. Best start floor = 4
-----------------------------------
Best floor was 4 with avg utility of -28.7686 over 1000 steps and 3 experiments.
```

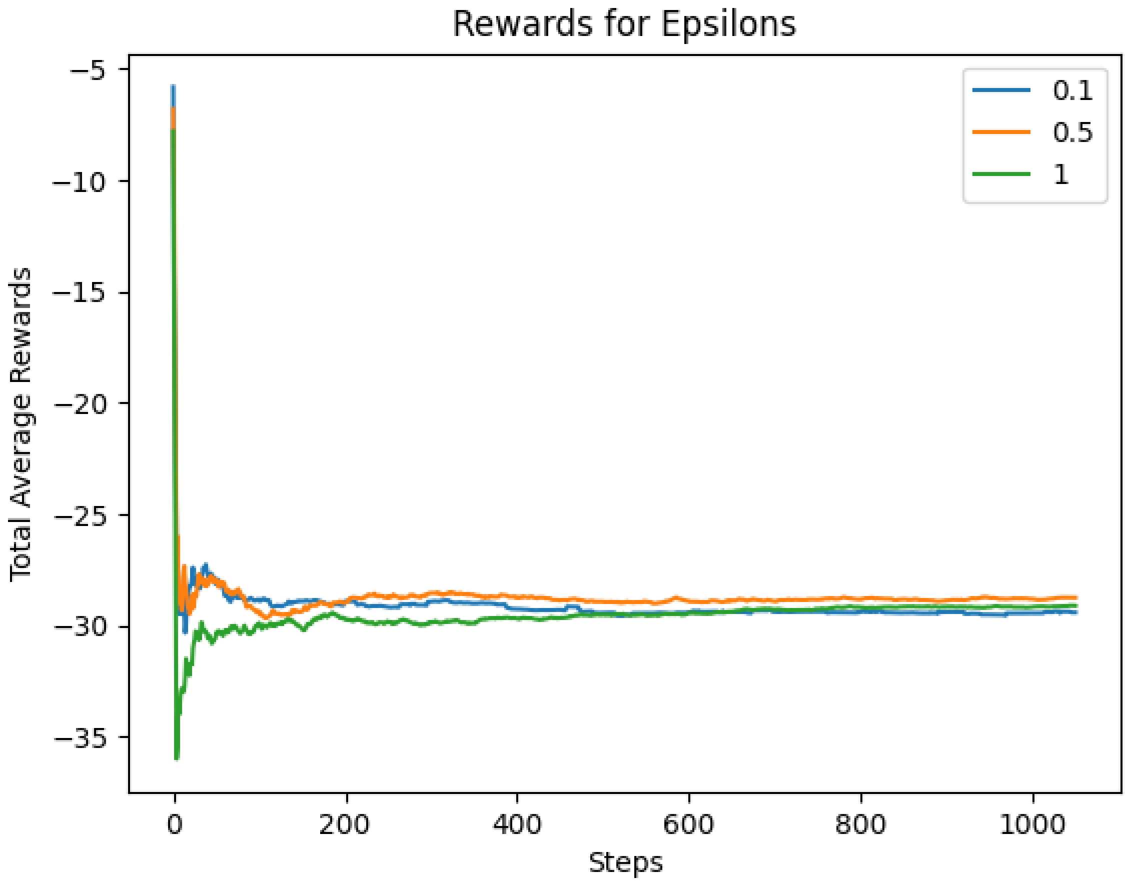
1.e Graph: Utilities ϵ = (1, .5, .1)

1.e Graph: Epsilon




Utility Over Time for Each Action
Action 2
Action 3
Action 4
Action 5
Action 6
−20.0
−22.5
−25.0
−27.5
−30.0
−32.5
−35.0
Utility [#]
0
100
200
300
400
500
Steps [#]


Utility Over Time for Each Action
Action 2
Action 3
Action 4
Action 5
Action 6
−20.0
−22.5
−25.0
−27.5
−30.0
−32.5
−35.0
−37.5
−40.0
Utility [#]
0
200
400
600
800
Steps [#]



Problem 1.e Explanations/Conclusions

The final eleavtor chosen was exclusively the new eleavtor of course, and it believed floor 4 was the best elevator. Better results can of course be obtained by increasing the steps and increasing the number and range of epsilon values.

Problem 2.a (Interpretation 1)

NOTE: The verbiage on this problem was also slightly confusing, so I ran the solution twice using two different interpretations. I use my second interpretaion on 2.b when rerunning 1.d and 1.e

Since the penalty becomes worse over time, we need to change the reward function. Each reward will become increasingly bigger and further deviate from the expected average.

Here is the quadratic reward function:

$$r((S_c),(S_e)) = max(-(5|(S_e)-(S_c)|+2(7))^2, -(7|(S_e)-(S_c)|+2(7))^2)$$

To measure the deviation, I created a loss function:

$$Loss = Loss(a) = \frac{1}{k+1}(r_{actual} - r_{predicted})$$

And here is the modified piece of code:

```
time_new = (5 * abs(s_c - s_e) + (2 * 7))**2
time_old = (7 * abs(s_c - s_e) + (2 * 7))**2
```

Here are the paramaters and output:

```
START_FLOORS = [1]
EXIT_FLOORS = [2,3,4,5,6]
START_PROB = [1]
EXIT_PROB = [.20, .20, .20, .20, .20]
VERBOSE = True
ACTIONS = [2, 3, 4, 5, 6]  # action choices
EXPERIMENTS = 10
STEPS = 2000
EPSILONS = [.1, .3, .5, .8, 1]
TITLE = "_2_a"

-----------------------------------
Epsilon (0.1) with a total average reward of -918.3196. Best start floor = 4
Epsilon (0.3) with a total average reward of -911.6434. Best start floor = 2
Epsilon (0.5) with a total average reward of -897.573. Best start floor = 4
Epsilon (0.8) with a total average reward of -907.4949. Best start floor = 4
Epsilon (1) with a total average reward of -900.4719. Best start floor = 4
-----------------------------------
Best floor was 4 with avg utility of -897.573 over 2000 steps and 5 experiments.
```

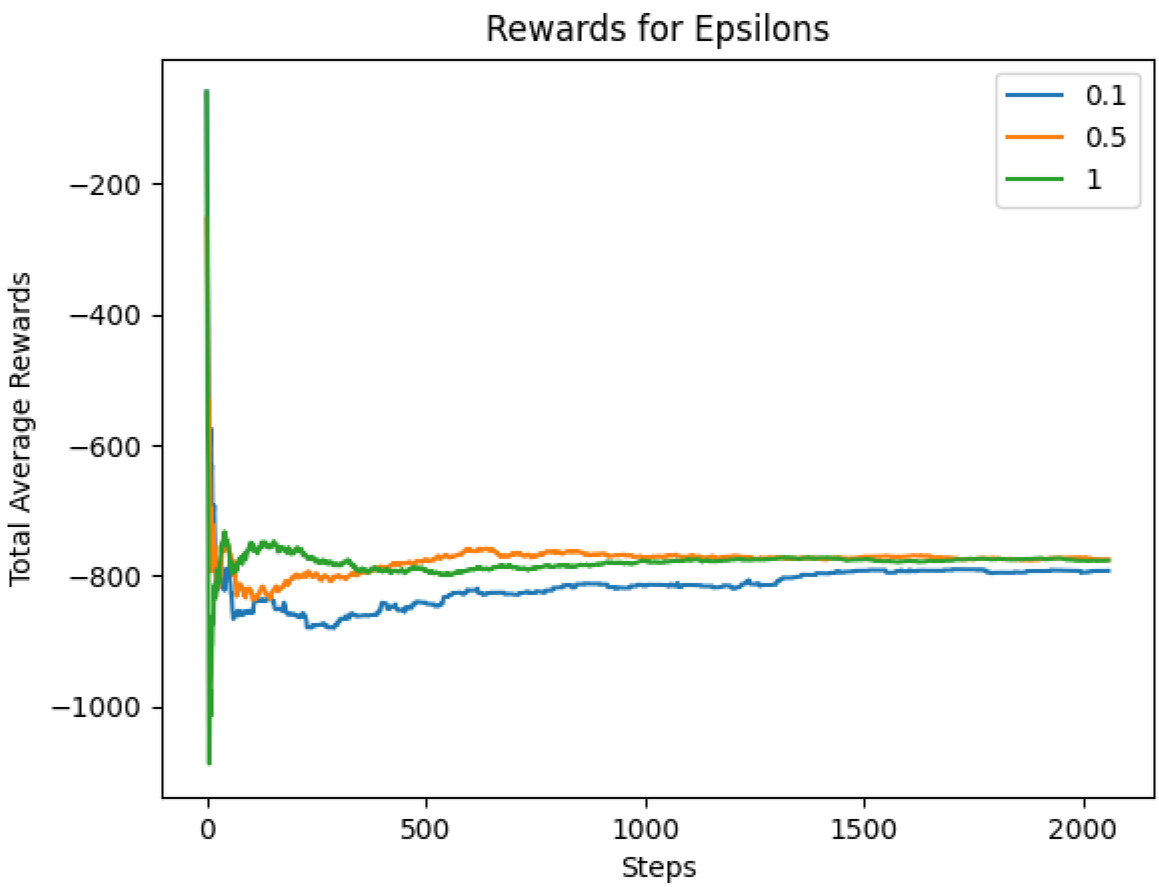
2.b (Interpretation 1)

For this part all we need to do is just add the new reward function made in part 2.a and run them with the code in 1.d and 1.e.

Here was the modified reward function:

```
time_new = (5 * abs(s_c - s_e) + (2 * 7))**2
time_old = (7 * abs(s_c - s_e) + (2 * 7))**2
```

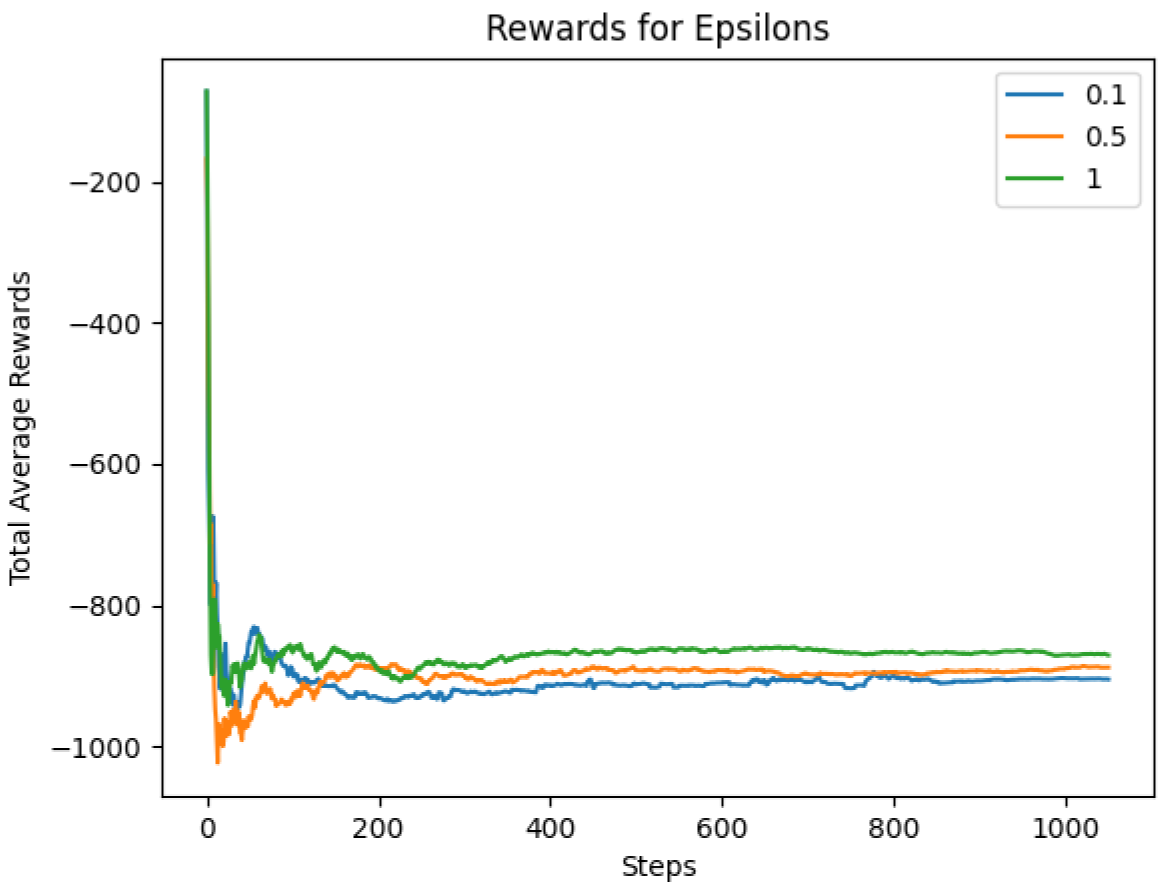
Epsilon graph for Rerunning 1.d



This is the output for running part d

```
-----------------------------------
Epsilon (0.1) with a total average reward of -815.2061. Best start floor = 4
Epsilon (0.5) with a total average reward of -775.822. Best start floor = 4
Epsilon (1) with a total average reward of -778.8782. Best start floor = 4
-----------------------------------
Best floor was 4 with avg utility of -775.822 over 2000 steps and 3 experiments.
```

Epsilon graph for Rerunning 1.e



This is the output for running part e

```
-----------------------------------
Epsilon (0.1) with a total average reward of -907.1112. Best start floor = 5
Epsilon (0.5) with a total average reward of -898.0325. Best start floor = 6
Epsilon (1) with a total average reward of -868.6103. Best start floor = 4
-----------------------------------
Best floor was 4 with avg utility of -868.6103 over 1000 steps and 3 experiments.
```

Problem 1.b Explanations/Conclusions

Compared to 1.d and 1.e, the results of the qaudratic function make the utility graph flucatuate more and the loss function longer. Additionaly, the utility beacuase of the quadratic reward. Additionally, had I used less steps, the qaudratic reward would not have had a completly different answer

Problem 2.b (Alternative 2)

The wording of problem 2.b was slightly ambigous so I ran the probelm again using a different interpretation. In this new interpretation, there is no average incrementatle update of the utility; the utility gets worse overtime.

I change this line in the utility:

```
self.q_val[action] = self.q_val[action] + (1 / (self.count[action] + 1) ) * (actual -
self.q_val[action])
```

to this:

```
self.q_val[action] = self.q_val[action] + (1 / (self.count[action] + 1) ) * (actual -
r_hat)
```

In particular, I changed

$$Q_k(a) = Q_k(a) + \frac{1}{k+1}(r(S_c, S_e)_{k+1} - Q_k(a))$$

to

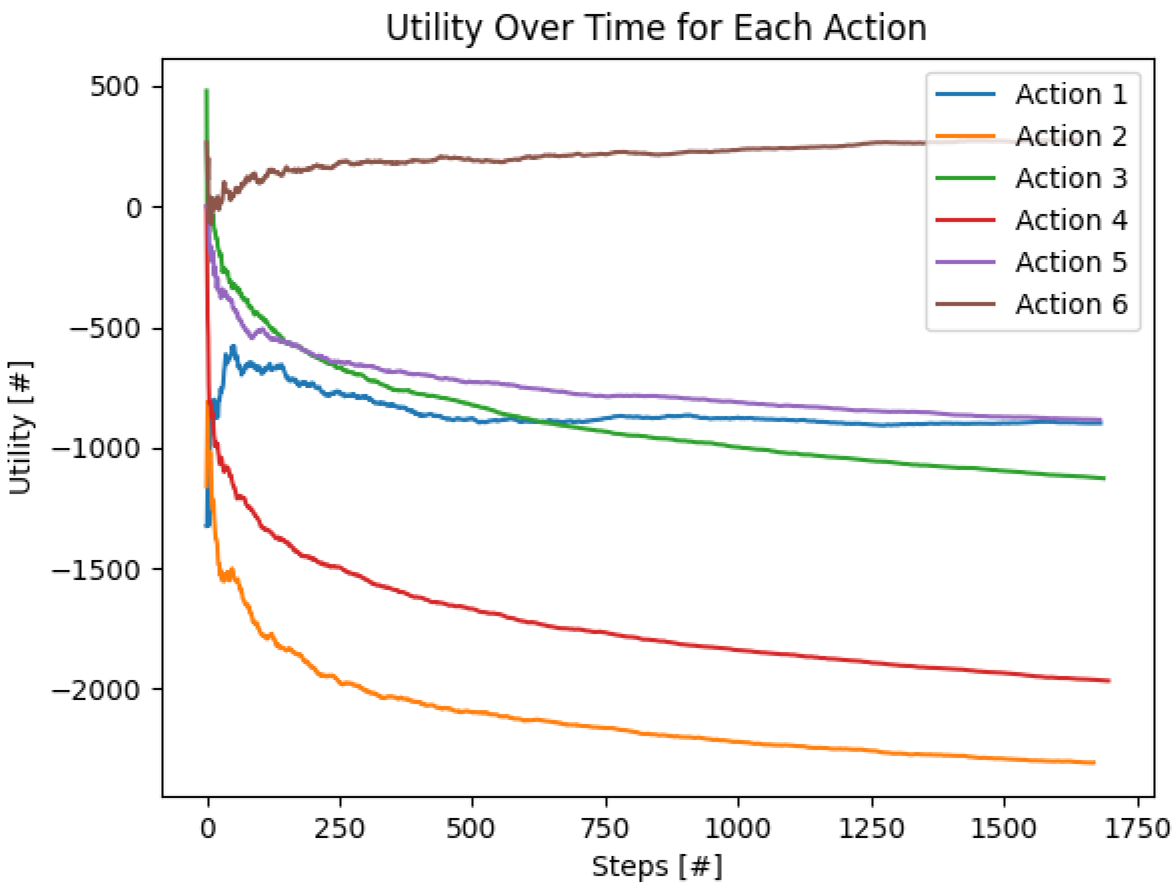
$$Q_k(a) = Q_k(a) + \frac{1}{k+1}(r(S_c, S_e)_{k+1} - r(a, S_e))$$

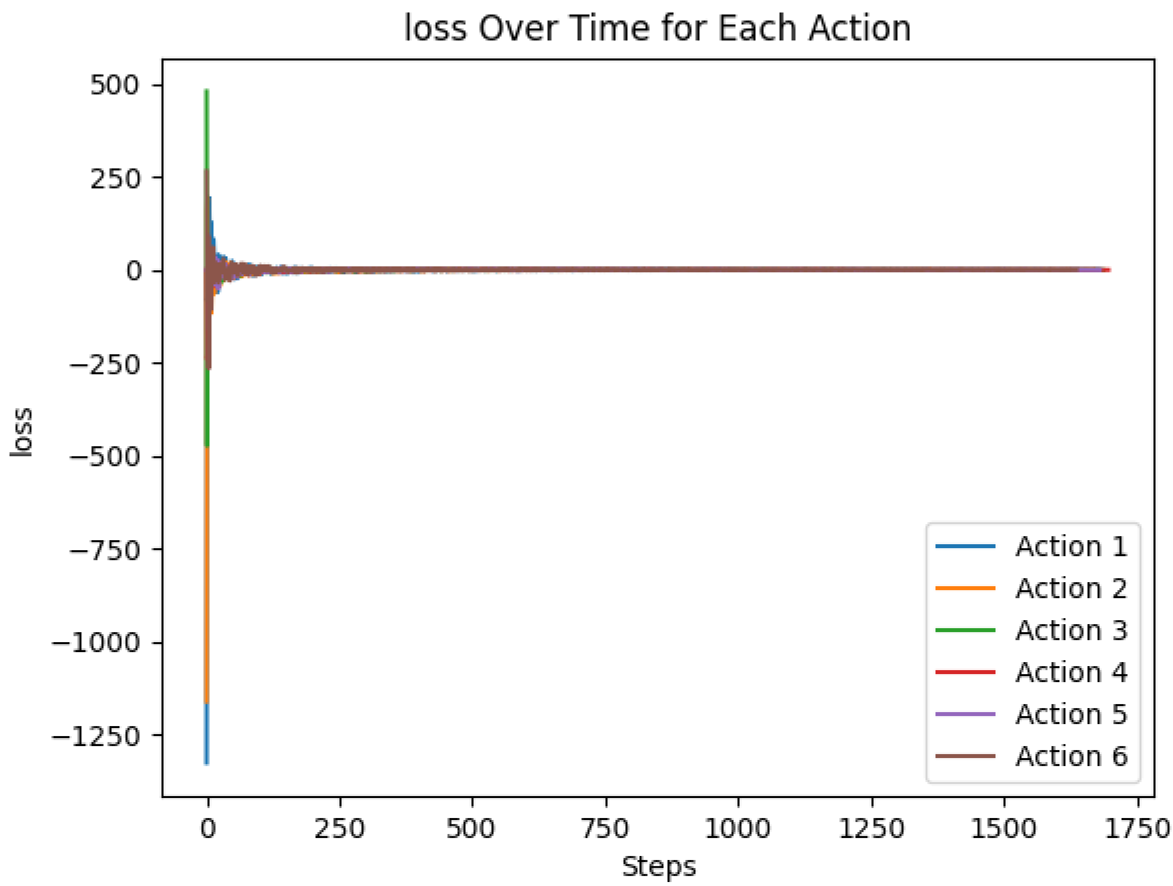
No longer was the utility being incrementally averaged. The new utility was added to the prediction error. Here are paramaters and output when running this with Part D:

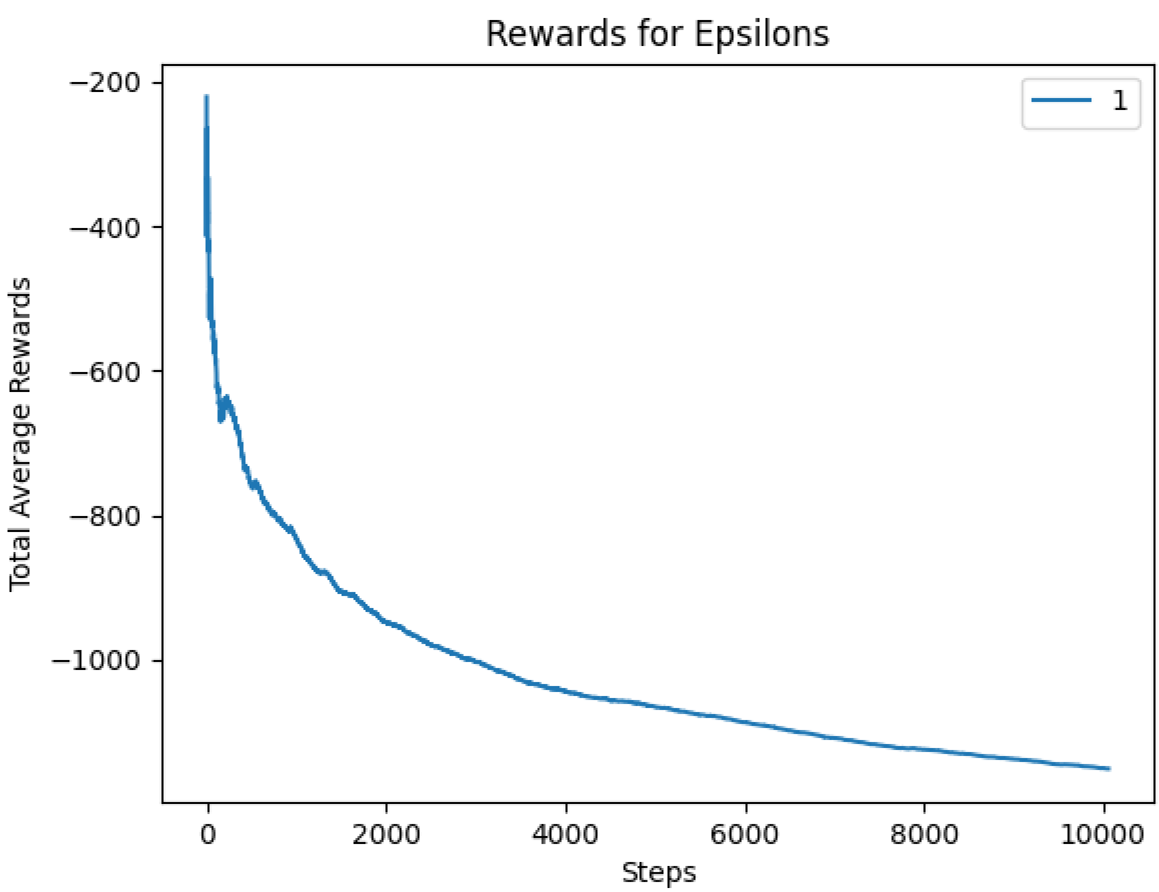
```
STEPS = 10000
EPSILONS = [1]
```

```
-----------------------------------
Action floor count = {1: 1683, 2: 1670, 3: 1689, 4: 1698, 5: 1682, 6: 1638}
Exit floor count = {0: 0, 1: 2631, 2: 1493, 3: 1111, 4: 1210, 5: 1587, 6: 2028}
Call floor count = {0: 0, 1: 1686, 2: 1684, 3: 1679, 4: 1670, 5: 1665, 6: 1676}
-----------------------------------
Epsilon (1) with a total average reward of -1024.5968. Best start floor = 6
-----------------------------------
Best floor was 6 with avg utility of -1024.5968 over 10000 steps and 1 experiments.
```

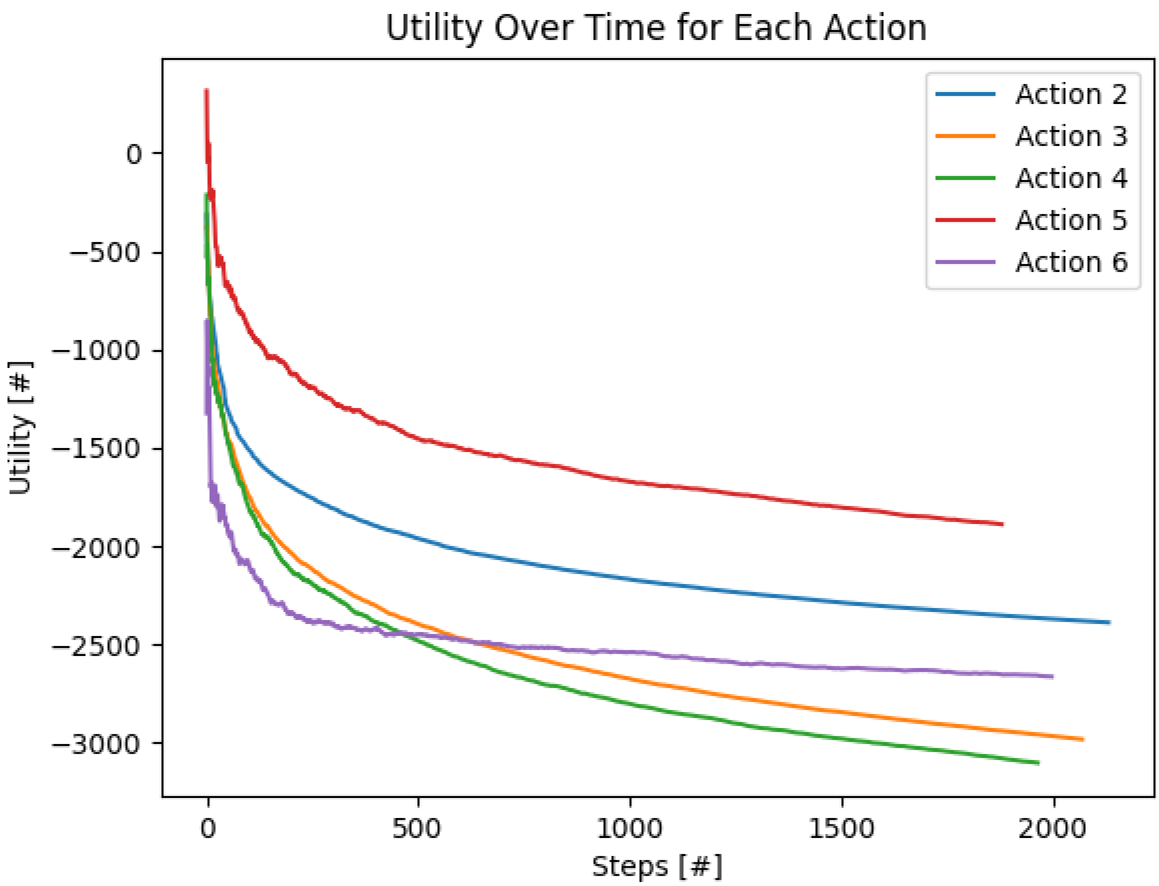
Utility, Loss, Epsilon Graph for 2.b Rerun of Part D (Alternative 2)

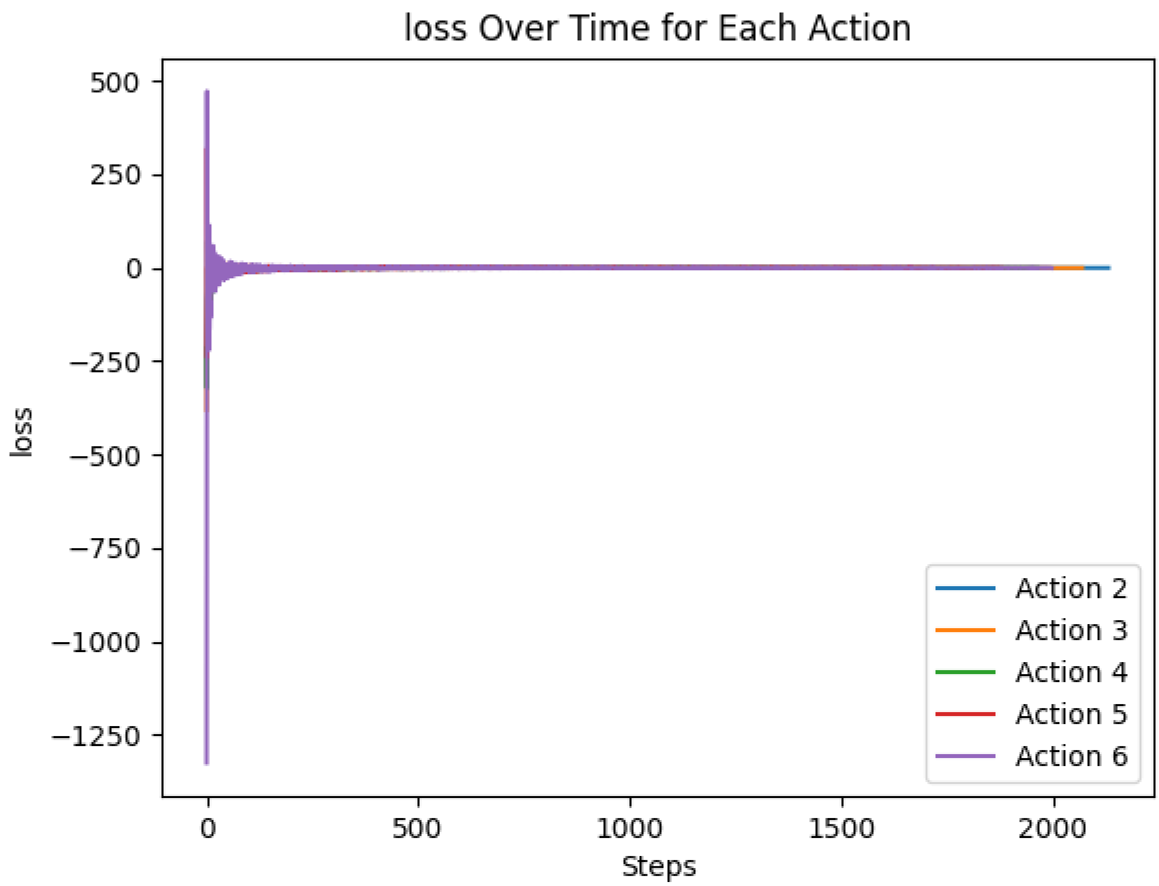



loss Over Time for Each Action
500
250
0
−250
−500
−750
−1000
−1250
loss
0
250
500
750
1000
1250
1500
1750
Steps
Action 1
Action 2
Action 3
Action 4
Action 5
Action 6

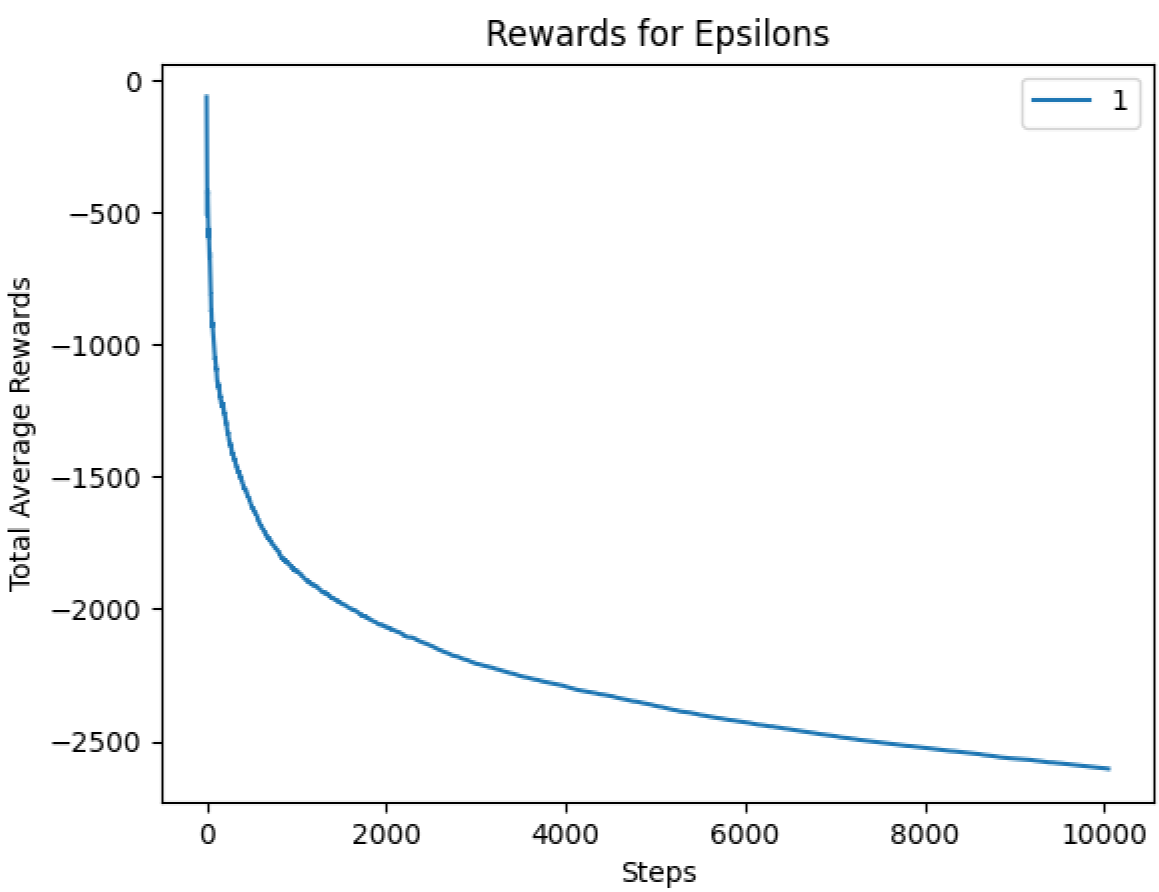

Rewards for Epsilons
−200
−400
−600
−800
−1000
Total Average Rewards
0
2000
4000
6000
8000
10000
Steps
1

Utility, Loss, Epsilon Graph for 2.b Rerun of Part E (Alternative 2)





loss Over Time for Each Action
500
250
0
−250
−500
−750
−1000
−1250
loss
0
500
1000
1500
2000
Steps
Action 2
Action 3
Action 4
Action 5
Action 6



Rewards for Epsilons
0
−500
−1000
−1500
−2000
−2500
Total Average Rewards
0
2000
4000
6000
8000
10000
Steps
1

Problem 1.b (Alternative 2) Explanations/Conclusions

For this alternative interpretation for problem 2.b.d we can see that the utilities follow a curve and no longer converge as with the incremental averaging solution. This leads to the utility getting worse and worse as time goes on.