CMSC 678 – Homework 4

Akshay Peshave

(peshave1@umbc.edu)

Question 1:

The intervals between the rewards hold significance. **The sign of the rewards ensure the presence of adequate intervals while making the reward structure more intuitive.** The discounted return for a state-action pair is given as:

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

If we add a constant C to all the rewards in the above equation we get

$$R_t' = \sum_{k=0}^{\infty} \gamma^k (r_{t+k+1} + C)$$
$$R_t' = \sum_{k=0}^{\infty} \left[\gamma^k r_{t+k+1} + \gamma^k C\right]$$
$$R_t' = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} + \sum_{k=0}^{\infty} \gamma^k C$$
$$R_t' = R_t + \sum_{k=0}^{\infty} \gamma^k C$$

We observe that adding a constant C to each step reward causes the discounted return to increase by a constant K, where

$$\boldsymbol{K = \sum_{k=0}^{\infty} \gamma^k C}$$

This is evidence that adding a constant value to the rewards does not affect the relative values of the states. This supports the earlier claim that intervals are more significant in the reward structure as opposed to the signs of individual rewards, which can be done away with by adding a constant value to all the rewards.

Question 3:

All Q-learning runs have been conducted using the ϵ-greedy approach. For all experiments the following parameters are constant:

Learning Rate, α = 0.1
Discount Rate, γ = 0.9
Sampled Episodes = 1,000

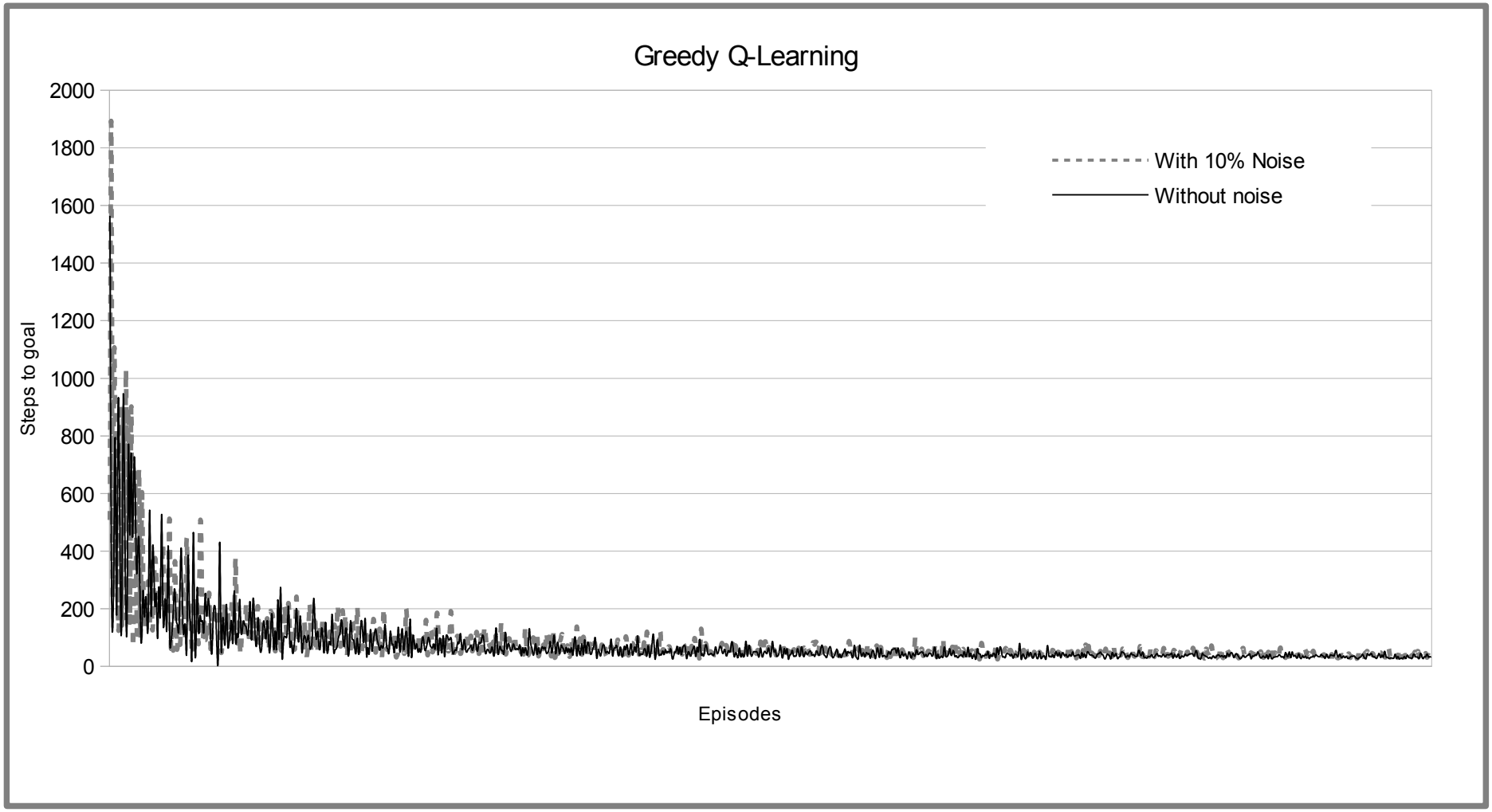
Initial experiment has been conducted with ϵ =0 i.e. the next step in each episode is chosen greedily based on expected return the move yields. ϵ = 0 implies random exploration is done only in case multiple moves yield the same expected return.

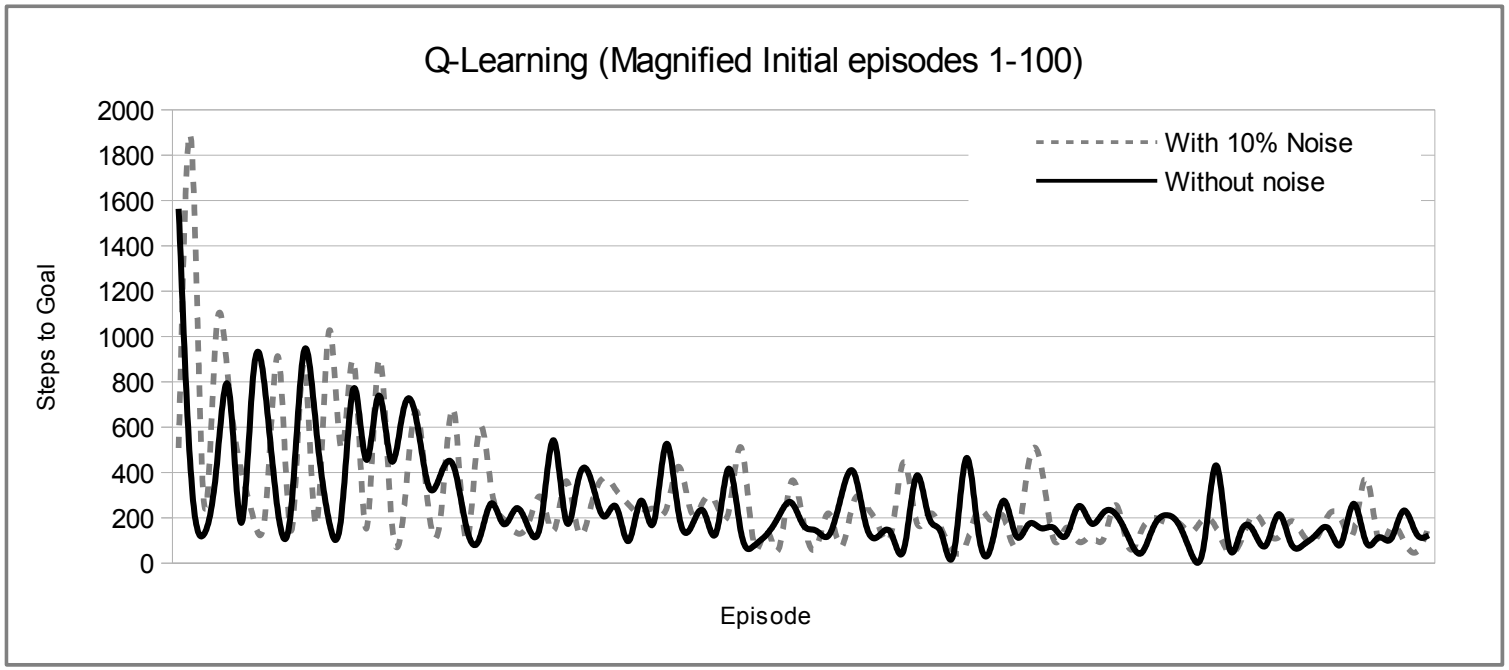
The second experiment has been conducted with ϵ = 0.1. This implies that 10% of the time the greedy decision based on expected return of a move is not followed and a move other than the greedy decision is randomly selected. This noise introduced in the greedy approach induces random exploration of the neighboring cells in the grid.

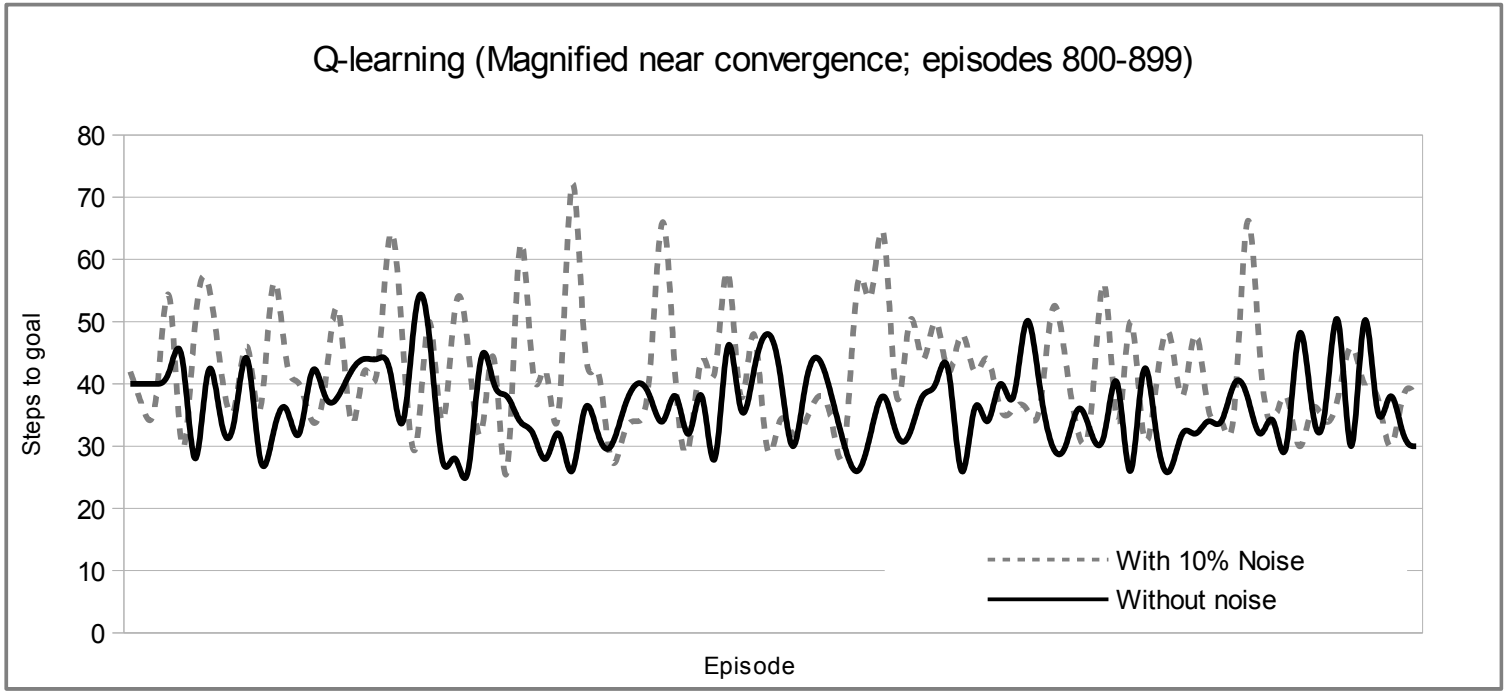
The learning curve for both experiments suggests that although both approaches converge the range of convergence is broader in case of the experiment with induced noise in the decision making. The optimal solution for a 15x15 grid is 28 steps.

In case of the pure greedy approach the learning algorithm converges in the range of 28-50`ish never settling on 28 since random stepping kicks in every time the Q-values for multiple moves converges on the optimal and mutually equal values. Steadily the new Q-values backup the grid and converge to the optimal relative intervals yielding the optimal value yet again. This beahvior is nearly periodic but does not exhibit symmetry due to random stepping.

In case of the noise-induced greedy approach the algorithm does converge but within a slightly broader range of 28-70`ish. This is due to noise which induces random movement on the grid irrespective of the optimal move indicated by the corresponding Q-values. Also we notice that this randomness causes the converging learning curve to spend lesser time around the optimal “28 steps” value and explore more than the vanilla greedy approach. This increased randomness causes a larger time-span to pass before the optimal relative intervals amongst Q-values are established.


Greedy Q-Learning
2000
1800
1600
1400
1200
1000
800
600
400
200
0
Steps to goal
Episodes
With 10% Noise
Without noise


Q-Learning (Magnified Initial episodes 1-100)
Steps to Goal
2000
1800
1600
1400
1200
1000
800
600
400
200
0
Episode
With 10% Noise
Without noise


Q-learning (Magnified near convergence; episodes 800-899)
Steps to goal
80
70
60
50
40
30
20
10
0
Episode
With 10% Noise
Without noise

```
import random


class QLearning:
    def __init__(self):

        self.gamma=0.9
        self.epsilon=0.9
        self.learningRate = 0.1
        self.gridWorld=[[[0.0, 0.0, 0.0, 0.0] for row in range(0,15)] for column in range(0,15)]

        self.maxEpisodes=1000

    def SelectEpsilonGreedyNeighbor(self, row, column):
        maxQ=-99999.99;
        maxQMove=[]
        possibleMoves=[]

        if row>0:
            possibleMoves.append(0)
            if maxQ<self.gridWorld[row][column][0]:
                maxQ=self.gridWorld[row][column][0]
                maxQMove=[0]
            elif maxQ==self.gridWorld[row][column][0]:
                maxQMove.append(0)

        if row<14:
            possibleMoves.append(1)
            if maxQ<self.gridWorld[row][column][1]:
                maxQ=self.gridWorld[row][column][1]
                maxQMove=[1]
            elif maxQ==self.gridWorld[row][column][1]:
                maxQMove.append(1)

        if column>0:
            possibleMoves.append(3)
            if maxQ<self.gridWorld[row][column][3]:
                maxQ=self.gridWorld[row][column][3]
                maxQMove=[3]
            elif maxQ==self.gridWorld[row][column][3]:
                maxQMove.append(3)

        if column<14:
            possibleMoves.append(2)
            if maxQ<self.gridWorld[row][column][2]:
                maxQ=self.gridWorld[row][column][2]
                maxQMove=[2]
```

```
        elif maxQ==self.gridWorld[row][column][2]:
            maxQMove.append(2)

    explorationProbability=random.randint(1,10)
    if explorationProbability/10.0 > self.epsilon:
        for move in maxQMove:
            possibleMoves.remove(move)

        if possibleMoves==[]:
            return [random.choice(maxQMove),maxQ]

        randomMove=random.choice(possibleMoves)

        if randomMove==0:
            QVal=self.gridWorld[row][column][0]
        elif randomMove==1:
            QVal=self.gridWorld[row][column][1]
        elif randomMove==2:
            QVal=self.gridWorld[row][column][2]
        else:
            QVal=self.gridWorld[row][column][3]
        return [randomMove,QVal]

    return [random.choice(maxQMove),maxQ]


def EpsilonGreedyLearn(self):
    episode=1
    while episode<=self.maxEpisodes:
        row=1
        column=1
        steps=0
        while True:
            nextState = self.SelectEpsilonGreedyNeighbor(row,column)

            steps+=1

            if nextState[0]==0:
                newRow= row-1
                newColumn=column
            elif nextState[0]==1:
                newRow=row+1
                newColumn=column
            elif nextState[0]==2:
                newColumn=column+1
                newRow=row
            else:
```

```
                    newColumn=column-1
                    newRow=row


                if newRow<0:
                    self.gridWorld[row][column][nextState[0]] = self.gridWorld[row][column][nextState[0]] + self.learningRate * -2
                elif newRow>14:
                    self.gridWorld[row][column][nextState[0]] = self.gridWorld[row][column][nextState[0]] + self.learningRate * -2
                elif newColumn<0:
                    self.gridWorld[row][column][nextState[0]] = self.gridWorld[row][column][nextState[0]] + self.learningRate * -2
                elif newColumn>14:
                    self.gridWorld[row][column][nextState[0]] = self.gridWorld[row][column][nextState[0]] + self.learningRate * -2
                else:
                    if newRow == 14 and newColumn==14:
                        self.gridWorld[row][column][nextState[0]] = self.gridWorld[row][column][nextState[0]] + self.learningRate *
(10 - self.gridWorld[row][column][nextState[0]])
                        break

                    futureMove=self.SelectEpsilonGreedyNeighbor(newRow,newColumn)

                    self.gridWorld[row][column][nextState[0]] = self.gridWorld[row][column][nextState[0]] + self.learningRate * (-1 +
(self.gamma*futureMove[1]) - self.gridWorld[row][column][nextState[0]])
                    row=newRow
                    column=newColumn
            print steps
            episode+=1

learner=QLearning()
learner.EpsilonGreedyLearn()
print learner.gridWorld
```

Question 3:

1)

$$P(A \wedge B \wedge C \wedge D \wedge E) = P(E \mid A \wedge B \wedge C \wedge D) * P(A \wedge B \wedge C \wedge D)$$
$$= P(E \mid C) * P(A \wedge B \wedge C \wedge D)$$
$$= P(E \mid C) * P(D \mid A \wedge B \wedge C) * P(A \wedge B \wedge C)$$
$$= P(E \mid C) * P(D \mid A \wedge B) * P(A \wedge B \wedge C)$$
$$= P(E \mid C) * P(D \mid A \wedge B) * P(C \mid A \wedge B) * P(A \wedge B)$$
$$= P(E \mid C) * P(D \mid A \wedge B) * P(C \mid A) * P(A \wedge B)$$
$$\mathbf{= P(E \mid C) * P(D \mid A \wedge B) * P(C \mid A) * P(A) * P(B)}$$

2)

$$P(C=T, D=F) = \begin{pmatrix} P(C=T, D=F \mid A=T, B=T) \\ + \\ P(C=T, D=F \mid A=T, B=F) \\ + \\ P(C=T, D=F \mid A=F, B=T) \\ + \\ P(C=T, D=F \mid A=F, B=F) \end{pmatrix}$$

$$= \begin{pmatrix} P(C=T \mid A=T) * P(D=F \mid A=T, B=T) * P(A=T) * P(B=T) \\ + \\ P(C=T \mid A=F) * P(D=F \mid A=T, B=F) * P(A=T) * P(B=F) \\ + \\ P(C=T \mid A=T) * P(D=F \mid A=F, B=T) * P(A=F) * P(B=T) \\ + \\ P(C=T \mid A=F) * P(D=F \mid A=F, B=F * P(A=F) * P(B=F) \end{pmatrix}$$

$$= \begin{pmatrix} 0.8*0.3*0.3*0.6 \\ + \\ 0.8*0.2*0.3*0.4 \\ + \\ 0.4*0.9*0.7*0.6 \\ + \\ 0.4*0.8*0.7*0.4 \end{pmatrix}$$

$$= \begin{pmatrix} 0.0432 \\ + \\ 0.1512 \\ + \\ 0.0192 \\ + \\ 0.0896 \end{pmatrix}$$

= 0.3032