# T.C.
# MARMARA UNIVERSITY
# FACULTY OF ENGINEERING
# COMPUTER ENGINEERING DEPARTMENT

CSE 4088
INTRODUCTION TO MACHINE LEARNING

PROJECT FINAL REPORT

Title of the Project
“Comparison of Reinforcement Learning Algorithms”

Group Members
150115045 Semiha Hazel Çavuş
150115051 Zeynep Kumralbaş

January 7, 2020

**Abstract**

Reinforcement learning (RL) is a machine learning technique where an agent takes an action in an environment, moves to the next state and receives rewards or punishments regarding this new state. So, it can learn a satisfactory (hopefully optimal or possibly a near optimal) policy that leads the agent to the goal state in the environment. RL is considered as a machine learning paradigm in addition to supervised and unsupervised learning. The agent uses its own experience to learn the environment that is unknown to it [1].

In this project Prioritized Sweeping that is model-based, Q-learning that is model-free and off-policy, Sarsa that is model-free and on-policy RL algorithms are implemented. RL is considered to work in grid environments in our scope. Then, the performances of these algorithms are compared.

# I. Overview

The subtasks that are accomplished during the project is given:
Subtask 1: Literature survey about RL and algorithms
Subtask 2: Implementation of Q-learning algorithm
Subtask 3: Prepare project midterm report
Subtask 4: Implementation of Sarsa algorithm
Subtask 5: Implementation of Prioritized Sweeping algorithm
Subtask 6: Comparing performances of these algorithms
Subtask 7: Preparing project final report and presentation
Each subtask is accomplished by all members equally.

# II. Project Accomplishment

## A. Research on Q-learning

Q-learning is model-free which can be thought as the agent does not have memory. The agent does not know the environment and, is not able to know.

Agent uses Q-values to learn. Q-values hold values for quality of actions since "Q" stands for quality. The agent can find out how valuable the given action is according to the action's Q-value. Q-table is a [ number of states , number of actions] matrix which holds Q-values for state, action pairs. The agent uses and updates these values from Q-table while it is learning.

Before learning starts, Q-table can be initialized arbitrarily or with zeros. Then the agent starts to interact with the environment at starting state. It knows current state, actions that it can take, and the states that are the result of the taken actions. The actions are chosen by a policy. For example, when $\epsilon$ -greedy action selection is used, with $\epsilon$ probability a random (explorative) action is selected and with $1-\epsilon$ probability action that has maximum Q-value (exploitative) is selected. To make the agent explorative at the beginning, $\epsilon$ is selected closer to 1. $\epsilon$ decreases as learning takes place thus, the agent starts to exploit Q-table. The agent selects actions based on the $\epsilon$ up to the terminating states (states that have goal or punishment). When the agent reaches the terminating states, an episode ends and a new episode starts at the starting state. Hopefully, the agent will learn the environment and convergence occurs to optimal Q-values ($q_*$)

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma max_a Q(\ S_{t+1}, A) - Q(S_t, A_t)]$$

The above formula is used to update Q-values after each action. $Q(S_t, A_t)$ is the Q-value for state $S$ and action $A$ at t. $\alpha$ stands for learning rate which is tuning parameter to determine the step size. $\gamma$ stands for discount factor which implies that importance of future reward to current state. $max_a Q(S_{t+1}, a)$ is the maximum Q-value from destination state and actions pairs.

There are two types of policies. Target policy (optimal policy) is the policy that the agent tries to learn. Behavior policy is the policy that defines agent's behavior. If the agent tries to learn the optimal policy but its behavior is exploratory, the aim of two kinds of policies does not match. It is called as off-policy. Let's think about Q-learning in this aspect. In the Q-value update rule, always the maximum Q-value is selected (target policy). If an exploitative action is selected, two policies match. But, if an explorative action is selected, two policies do not match. Therefore, Q-learning is an off-policy algorithm.

**Algorithm 1** : Q-Learning Algorithm

**Algorithm parameters:** step size $\alpha \in (0, 1]$, small $\epsilon > 0$

Initialize $Q(s, a)$, for all s $\in S^{+}$, $a \in A(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$
**for** each episode **do**
    Initialize S
    **for** each step of episode: **do**
        Choose $A$ from $S$ using policy derived from $Q(e.g., \epsilon - greedy)$
        Take action $A$, observe $R, S^{'}$
        $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma max_a Q( S^{'} , A) - Q(S, A)]$
        $S \leftarrow S^{'}$
    **end for**
    until $S$ is terminal
**end for**

### B. *Implementation of Q-learning*

A grid environment consists of normal states, goal and punishment states. Each state has left, right, up and down actions, but the border states of the grid environment have limited valid actions. Learning parameters of the RL algorithm are given before running the code.

When the RL task is run, the agent starts to explore and learn the environment. It chooses an action based on $\epsilon$-greedy policy, an explorative action with $\epsilon$ probability and an exploitative action with $1-\epsilon$ probability. An explorative action corresponds to a random action, whereas an exploitative action corresponds to the action that has maximum Q. Each Q value of the state-action pairs are updated in each step. When the agent reaches an terminating state, goal or punishment state, one episode finishes. Then, the agent starts to explore the environment from its starting coordinates in a new episode.

The learning process continues until the optimal path is found. The variance of the number of actions in the last N steps (N is decided by the user) are computed to determine whether the learning is occured or not. If the variance becomes zero, we can say that the agent learned the environment and found the optimal path.

After each episode, the mean values of the number of actions taken so far is computed. The mean values of number of actions versus number of episodes graph is plotted. This graph can be used as a property to compare the RL algorithms. The graph that is the output of an example run of the RL task, where the environment size is $5 \times 5$, $\alpha = 0.2$, $\gamma = 0.9$, $\epsilon = 0.35$, convergeInterval $= 10$, reward$= 500$, punishment $= -500$, is given in Fig. 1.

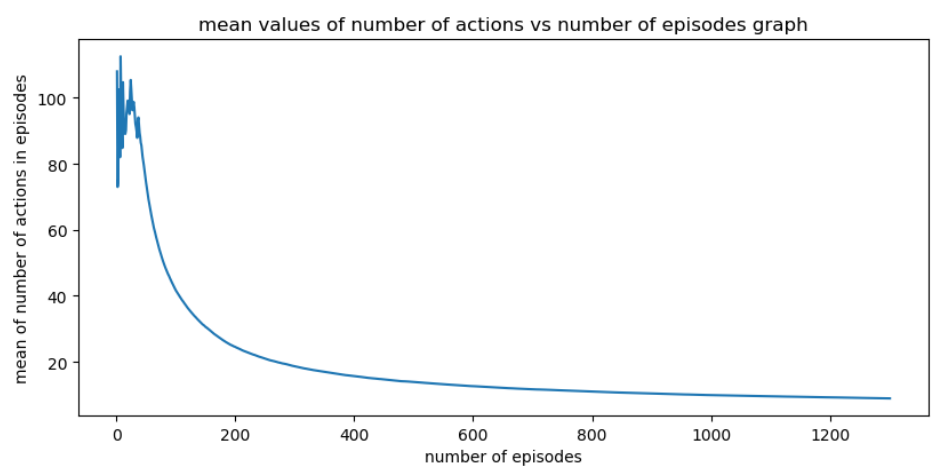


Figure 1: mean values of number of actions vs number of episodes graph

### C. SARSA Algorithm

SARSA is another model-free RL algorithm. But it is an on-policy algorithm. The SARSA algorithm is very similar to Q-Learning algorithm except that it uses the same policy for target and behavior policies. While updating the Q-value, it gets the Q-Value of an action with $\epsilon$-greedy policy in the destination state.

---

**Algorithm 2** : SARSA Algorithm

---

**Algorithm parameters:** step size $\alpha \in (0, 1]$, small $\epsilon > 0$

Initialize $Q(s, a)$, for all s $\in S^{+}$, $a \in A(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$
**for** each episode **do**
    Initialize S
    **for** each step of episode: **do**
        Choose $A$ from $S$ using policy derived from $Q(e.g., \epsilon - greedy)$
        Take action $A$, observe $R, S^{'}$
        $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(\ S^{'}\ , A) - Q(S, A)]$
        $S \leftarrow S^{'}$
    **end for**
    until $S$ is terminal
**end for**

---

### D. Q-Learning vs SARSA

Q-Learning and SARSA are run on a cliff walking environment in Fig. 2. The states which are the cliff have a punishment -100. The goal state has reward with value of 500. The rewards of other states are -1.

The parameters for this experiment as follows:

$\alpha$ = 0.1, $\gamma$ = 0.9, $\epsilon$ = 0.1, $\epsilon_{min}$ = 0.00001, $\epsilon$ decay = 0.00001
Convergence criteria = loop until variance is zero or current $\epsilon$ is equal to $\epsilon_{min}$.

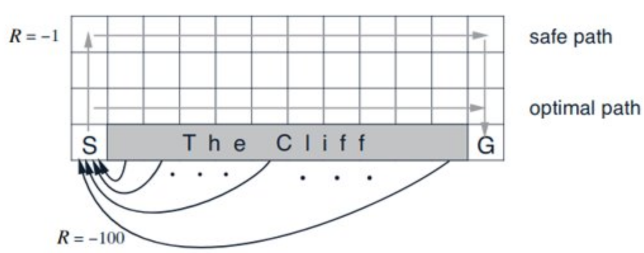


Figure 2: An Example: Cliff Walking

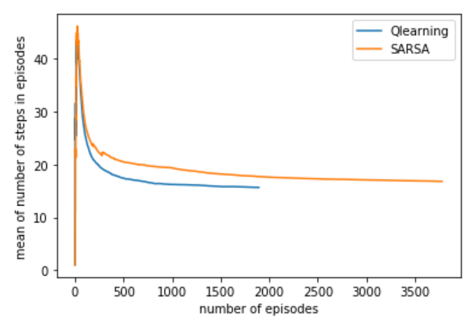


Figure 3: mean values of number of actions vs number of episodes graph

While the agent is learning the environment in SARSA, if it falls from the cliff, the Q-values of states that are around the cliff decrease more than in the Qlearning. So, agent avoids going these states and tries to find a safe path. Whereas in the Q-learning, when states are updated around the cliff, Q-values are updated using the action which gives the max Q-value. So, the agent does not avoid walking around the cliff.

The graph in Fig. 3 shows that Qlearning finds a shorter path than SARSA and converges faster.

### E. Prioritized Sweeping Algorithm (PS)

**Algorithm 3** : Prioritized Sweeping Algorithm

Initialize $Q(s,a)$, $Model(s,a)$ and $PQueue$ to empty
**loop** forever
  $s \leftarrow current$ (non-terminal) $state$
  $a \leftarrow policy(s,Q)$
  Execute a; observe s' and r
  $Model(s,a) \leftarrow (s',r)$ //assuming deterministic environments
  $p \leftarrow |r + \gamma max_{a'} Q(s',a') – Q(s,a)|$
  **if** $(p > \theta)$ **then**
    Insert $(s,a)$ into $PQueue$ with priority $p$
  **end if**
  **for** $N$ times while $PQueue$ is not empty **do**
    $(s,a) \leftarrow$ first($PQueue$)
    $(s',r) \leftarrow Model(s,a)$
    $Q(s,a) = Q(s,a) + \alpha[r + \gamma max_{a'} Q(s',a') - Q(s,a)]$
    **for** all $s'', a''$ (from within all previously experienced pairs) predicted to lead to $s$ **do**
      $r'' \leftarrow$ predicted reward
      $p \leftarrow r'' + \gamma max_{a'} Q(s,a) – Q(s'',a'')$
      **if** $p > \theta$ **then**
        insert $(s'',a'')$ into $PQueue$ with priority $p$
      **end if**
    **end for**
  **end for**
**end loop**

Agent memorizes its experienced states and rewards, inserts into a model. So it memorizes the environment.

Using the model, while it is updating the state-action pairs, it updates the N state-action pairs that it has saved to the model.To make more efficient selections from the model, p value and a priority queue is used. p value is the update part of the Q-value, it helps to avoid selecting state-action pairs which have nearly no effect of update part.If p is bigger than the threshold $\theta$, state-action pair is inserted to a priority queue.

Instead of taking state-action pairs from the model, they are taken from the priority queue. Hence the ones that have higher Q-values are updated more frequently, and learning converges faster.

### F. Q-Learning vs PS

Q-Learning and PS are run on a 7x7 grid environment in Fig. 4. There is a punishment with value of -500. The goal state has reward with value of 500.

The parameters for this experiment as follows:
$\alpha$ = 0.1, $\gamma$ = 0.9, $\epsilon$ = 0.7, $\epsilon_{min}$ = 0.00001, $\epsilon$ decay = 0.001, N = 5, $\theta$ = 0.04
Convergence criteria = loop until variance is zero or current $\epsilon$ is equal to $\epsilon_{min}$.

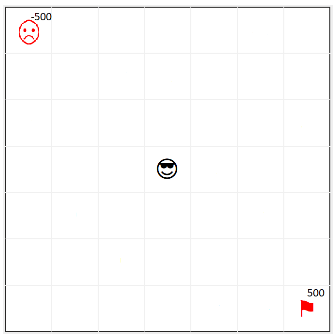


Figure 4: An Example: 7x7 grid world

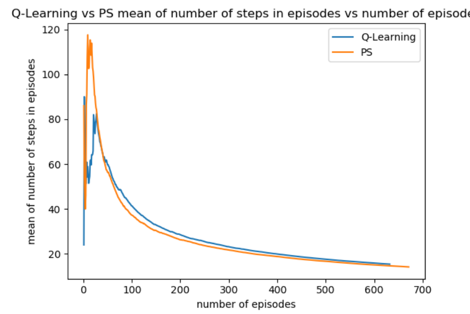


Figure 5: mean values of number of actions vs number of episodes graph

PS is converged faster than the Q-Learning. The differences of the convergence times between PS and Q-Learning is not seen clearly in Fig. 5. If the state space is large, we expect to see differences more clearly.

## III. Summary

To sum up what we learned from this project, all three algorithms can be used for different purposes. For example, if we want the solution to be more conservative, SARSA algorithm should be considered. If the risks are not too serious, Q-learning can be a better choice to find a path that is hopefully optimal or nearly optimal path.
PS algorithm converges faster than SARSA or Q-learning. The advantage of PS over the others cannot be observed on small environments, although on large environments, using the PS algorithm is more reasonable since PS has advantage on convergence.

## Appendix A

### A. main.py

```python
from RLTask import RLTask
from PS import PS
import numpy as np
import matplotlib.pyplot as plt

# =======Q-LEARNING VS PS==================================================================

RLTask_Qlearning = RLTask((7, 7))
RLTask_Qlearning.applyQLearning()

episodesForQlearning = np.arange(1, RLTask_Qlearning.numOfEpisodes+1)  # [1, numOfEpisodes+1)
plt.title("Q-Learning vs PS mean of number of steps in episodes vs number of episodes")
plt.xlabel("number of episodes")
plt.ylabel("mean of number of steps in episodes")
plt.plot(episodesForQlearning, RLTask_Qlearning.meanValues)
print("Qlearning total # of episodes:", RLTask_Qlearning.numOfEpisodes)

PS_task = PS((7, 7))
PS_task.applyQLearningPS()
episodesForPS = np.arange(1, PS_task.numOfEpisodes+1)  # [1, numOfEpisodes+1)
plt.plot(episodesForPS, PS_task.meanValues)
plt.legend(['Q-Learning', 'PS'], loc='upper right')
print("PS total # of episodes:", PS_task.numOfEpisodes)
plt.show()

# =======Q-LEARNING VS PS==================================================================
```

```python

# =======Q-LEARNING VS SARSA==================================================================
RLTask_Qlearning = RLTask((5, 5))
RLTask_Qlearning.applyQLearning()

episodesForQlearning = np.arange(1, RLTask_Qlearning.numOfEpisodes+1)  # [1, numOfEpisodes+1)
plt.title("Q-Learning␣vs␣PS␣mean␣of␣number␣of␣steps␣in␣episodes␣vs␣number␣of␣episodes")
plt.xlabel("number␣of␣episodes")
plt.ylabel("mean␣of␣number␣of␣steps␣in␣episodes")
plt.plot(episodesForQlearning, RLTask_Qlearning.meanValues)
plt.show()

RLTask_SARSA = RLTask((5, 5))
RLTask_SARSA.applySARSA()
episodesForSARSA = np.arange(1, RLTask_SARSA.numOfEpisodes+1)  # [1, numOfEpisodes+1)
plt.title("SARSA␣mean␣of␣number␣of␣steps␣in␣episodes␣vs␣number␣of␣episodes")
plt.xlabel("number␣of␣episodes")
plt.ylabel("mean␣of␣number␣of␣steps␣in␣episodes")
plt.plot(episodesForSARSA, RLTask_SARSA.meanValues)
plt.show()

print("Qlearning␣total␣#␣of␣episodes:", RLTask_Qlearning.numOfEpisodes)
print("SARSA␣total␣#␣of␣episodes:", RLTask_SARSA.numOfEpisodes)


# =======Q-LEARNING VS SARSA==================================================================
```

Listing 1: Python - main.py

### *B. PQueue.py*

```python
import heapq

from queue import PriorityQueue
pq = PriorityQueue()

class PQueue(object):

    def __init__(self):
        self.queue = []

    def __str__(self):
        return '␣'.join([str(i) for i in self.queue])

        # for checking if the queue is empty

    def isEmpty(self):
        return len(self.queue) == 0  # []

        # for inserting an element in the queue

    def insert(self, theta, tuple):

        self.queue.append([theta, tuple])

        # for popping an element based on Priority
    def delete(self):
        try:
            maxx = 0
            for i in range(len(self.queue)):
                if self.queue[i][0] > self.queue[maxx][0]:
                    maxx = i
            item = self.queue[maxx][1]
            del self.queue[maxx]
            return item
        except IndexError:
            print()
            exit()
'''
if __name__ == '__main__':
    myQueue = PQueue()
    myQueue.insert(5, (2, 5))
    myQueue.insert(3, (1, 3))
    a = myQueue.delete()
```

```
    print(a)
#   myQueue.insert(14)
  #  myQueue.insert(7)
'''
```

Listing 2: Python - PQueue.py

### C. RLTask.py

```
from statistics import variance
import numpy as np

from Environment import Environment


class RLTask:

    def __init__(self, size):

        self.__environmentSize = size
        self.__environment = Environment(self.__environmentSize)
        # parameters
        self.__alpha = 0.1
        self.__gamma = 0.90
        self.__maxEpsilon = 0.7
        self.__minEpsilon = 0.00001
        self.__epsilonDecrease = 0.001
        self.__currentEpsilon = self.__maxEpsilon
        self.__goalReward = 50
        self.__actionDictionary = {"LEFT": "1", 'UP': "2", "RIGHT": "3", "DOWN": "4"}
        self.__meanValues = []
        self.__numOfEpisodes = 0

        self.__totalReward = 0
        self.__totalRewardArray = []

        # convergence parameters
        self.__convergenceInterval = 10

        self.__lastNSteps = np.zeros(self.__convergenceInterval)  # create array for converge interval

    @property
    def environmentSize(self):
        return self.__environmentSize

    @property
    def environment(self):
        return self.__environment

    @property
    def alpha(self):
        return self.__alpha

    @alpha.setter
    def alpha(self, alpha):
        self.__alpha = alpha

    @property
    def gamma(self):
        return self.__gamma

    @gamma.setter
    def gamma(self, gamma):
        self.__gamma = gamma

    @property
    def maxEpsilon(self):
        return self.__maxEpsilon

    @maxEpsilon.setter
    def maxEpsilon(self, maxEpsilon):
        self.__maxEpsilon = maxEpsilon

    @property
```

```
    def minEpsilon(self):
        return self.__minEpsilon

    @minEpsilon.setter
    def minEpsilon(self, minEpsilon):
        self.__minEpsilon = minEpsilon

    @property
    def epsilonDecrease(self):
        return self.__epsilonDecrease

    @epsilonDecrease.setter
    def epsilonDecrease(self, epsilonDecrease):
        self.__epsilonDecrease = epsilonDecrease

    @property
    def goalReward(self):
        return self.__goalReward

    @goalReward.setter
    def goalReward(self, goalReward):
        self.__goalReward = goalReward

    @property
    def actionDictionary(self):
        return self.__actionDictionary

    @property
    def numOfEpisodes(self):
        return self.__numOfEpisodes

    def setEpsilonDecrease(self, epsilonDecrease):
        self.__epsilonDecrease = epsilonDecrease

    def setGoalReward(self, goalReward):
        self.__goalReward = goalReward

    def setConvergenceInterval(self, convergenceInterval):
        self.__convergenceInterval = int(convergenceInterval)

    @property
    def meanValues(self):
        return self.__meanValues

    @property
    def totalRewardArray(self):
        return self.__totalRewardArray

    def applyQLearning(self):

        stateList = self.__environment.stateList

        numOfActionsInEpisode = totalNumOfActions = 0
        self.__currentEpsilon = self.__maxEpsilon

        var = 1

        # episode loop
        while var != 0 and self.__currentEpsilon > self.__minEpsilon:

            agentCurrentCoordinates = self.__environment.startCoordinates  # initial coordinates of the agent
            self.__numOfEpisodes += 1
            numOfActionsInEpisode = 0

            # action loop
            currentState = stateList[agentCurrentCoordinates]

            # 1 episode finishes when the agent reaches the goal state or the punishment state
            while not (currentState.isGoal or currentState.isPunishment):
                currentState = stateList[agentCurrentCoordinates]

                chosenAction = currentState.choseAction(self.__currentEpsilon)  # chose an action based on epsilon

                # get the destination coordinates
                destinationCoordinates = self.__environment.getDestState(agentCurrentCoordinates,
```

```
chosenAction)

                destinationState = stateList[destinationCoordinates]  # get the destination state

                # update the QValue
                currentState.setQValues(chosenAction, currentState.getQValues(chosenAction) + self.__alpha
* (
                        destinationState.reward + self.__gamma * destinationState.getMaxQValue() -
                        currentState.getQValues(chosenAction)))

                self.__totalReward += destinationState.reward

                agentCurrentCoordinates = destinationCoordinates
                numOfActionsInEpisode += 1

                currentState = destinationState

            # 1 episode finishes

            self.__totalRewardArray.append(self.__totalReward)

            print("Qlearning currentEpsilon:", self.__currentEpsilon)

            # convergence calculation

            # if the lastNSteps array is full, shift the array and get the new value
            if self.__numOfEpisodes > self.__convergenceInterval:
                self.__lastNSteps[0:self.__convergenceInterval - 1] = self.__lastNSteps[1:]
                self.__lastNSteps[self.__convergenceInterval - 1] = numOfActionsInEpisode

            # if the lastNSteps array is not full, get the values
            else:
                self.__lastNSteps[self.__numOfEpisodes - 1] = numOfActionsInEpisode

            totalNumOfActions += numOfActionsInEpisode

            # mean values of the number of actions in episodes calculation

            if len(self.__meanValues) == 0:
                self.__meanValues.append(numOfActionsInEpisode / self.__numOfEpisodes)
            else:
                lastMean = self.__meanValues[-1]
                self.__meanValues.append(
                    (lastMean * (self.__numOfEpisodes - 1) + numOfActionsInEpisode) / self.__numOfEpisodes)

            self.__currentEpsilon -= self.__epsilonDecrease
            var = variance(self.__lastNSteps)
            print("Qlearning variance:", var)


        # print Qvalues
        self.printQValues(stateList, "Qlearning")

    def applySARSA(self):

        stateList = self.__environment.stateList

        numOfActionsInEpisode = totalNumOfActions = 0
        self.__currentEpsilon = self.__maxEpsilon

        var = 1

        # episode loop
        while var != 0 and self.__currentEpsilon > self.__minEpsilon:

            agentCurrentCoordinates = self.__environment.startCoordinates  # initial coordinates of the
agent
            self.__numOfEpisodes += 1
            numOfActionsInEpisode = 0

            # action loop
            currentState = stateList[agentCurrentCoordinates]

            # 1 episode finishes when the agent reaches the goal state or the punishment state
            while not (currentState.isGoal or currentState.isPunishment):
                currentState = stateList[agentCurrentCoordinates]

```

```python
            chosenAction = currentState.choseAction(self.__currentEpsilon)  # chose an action based on epsilon

            # get the destination coordinates
            destinationCoordinates = self.__environment.getDestState(agentCurrentCoordinates, chosenAction)

            destinationState = stateList[destinationCoordinates]  # get the destination state

            chosenAction_t1 = destinationState.choseAction(self.__currentEpsilon)  # get at+1

            # update the QValue
            currentState.setQValues(chosenAction, currentState.getQValues(chosenAction) + self.__alpha * (
                    destinationState.reward + self.__gamma * destinationState.getQValues(chosenAction_t1) -
                    currentState.getQValues(chosenAction)))

            self.__totalReward += destinationState.reward

            agentCurrentCoordinates = destinationCoordinates
            numOfActionsInEpisode += 1

            currentState = destinationState

        # 1 episode finishes

        self.__totalRewardArray.append(self.__totalReward)

        print("SARSA currentEpsilon:", self.__currentEpsilon)

        # convergence calculation

        # if the lastNSteps array is full, shift the array and get the new value
        if self.__numOfEpisodes > self.__convergenceInterval:
            self.__lastNSteps[0:self.__convergenceInterval - 1] = self.__lastNSteps[1:]
            self.__lastNSteps[self.__convergenceInterval - 1] = numOfActionsInEpisode

        # if the lastNSteps array is not full, get the values
        else:
            self.__lastNSteps[self.__numOfEpisodes - 1] = numOfActionsInEpisode

        totalNumOfActions += numOfActionsInEpisode

        # mean values of the number of actions in episodes calculation

        if len(self.__meanValues) == 0:
            self.__meanValues.append(numOfActionsInEpisode / self.__numOfEpisodes)
        else:
            lastMean = self.__meanValues[-1]
            self.__meanValues.append(
                (lastMean * (self.__numOfEpisodes - 1) + numOfActionsInEpisode) / self.__numOfEpisodes)

        self.__currentEpsilon -= self.__epsilonDecrease
        var = variance(self.__lastNSteps)
        print("SARSA variance:", var)

    # print Qvalues
    self.printQValues(stateList, "SARSA")

def printQValues(self, stateList, method):
    print(method)
    for i in range(0, self.__environmentSize[0]):
        for j in range(0, self.__environmentSize[1]):
            print(i, j)
            if 'L' in stateList[i][j].validActions:
                print("L:", stateList[i][j].getQValues("L"), " ")
            if 'R' in stateList[i][j].validActions:
                print("R:", stateList[i][j].getQValues("R"), " ")
            if 'U' in stateList[i][j].validActions:
                print("U:", stateList[i][j].getQValues("U"), " ")
            if 'D' in stateList[i][j].validActions:
                print("D:", stateList[i][j].getQValues("D"), " ")
```

Listing 3: Python - RLTask.py

### *D. PS.py*

```
from statistics import variance
import numpy as np
from State import State
from Environment import Environment
from PQueue import PQueue


class PS:

    def __init__(self, size):

        self.__environmentSize = size
        self.__modelSize = size[0] * size[1]
        self.__Model = [[(-1, -1)] * 4 for _ in range(self.__modelSize)]  # 4 for numOfActions
        self.__environment = Environment(self.__environmentSize)
        # parameters
        self.__alpha = 0.1
        self.__gamma = 0.90
        self.__theta = 0.04
        self.__N = 5
        self.__maxEpsilon = 0.7
        self.__minEpsilon = 0.00001
        self.__epsilonDecrease = 0.001
        self.__currentEpsilon = self.__maxEpsilon
        self.__goalReward = 50
        self.__actionDictionary = {"L": 1, 'U': 2, "R": 3, "D": 4}
        self.__meanValues = []
        self.__numOfEpisodes = 0

        self.__PQueue = PQueue()

        self.__totalReward = 0
        self.__totalRewardArray = []

        # convergence parameters
        self.__convergenceInterval = 10

        self.__lastNSteps = np.zeros(self.__convergenceInterval)  # create array for converge interval

    @property
    def environmentSize(self):
        return self.__environmentSize

    @property
    def environment(self):
        return self.__environment

    @property
    def alpha(self):
        return self.__alpha

    @alpha.setter
    def alpha(self, alpha):
        self.__alpha = alpha

    @property
    def gamma(self):
        return self.__gamma

    @gamma.setter
    def gamma(self, gamma):
        self.__gamma = gamma

    @property
    def maxEpsilon(self):
        return self.__maxEpsilon

    @maxEpsilon.setter
    def maxEpsilon(self, maxEpsilon):
        self.__maxEpsilon = maxEpsilon

    @property
    def minEpsilon(self):
        return self.__minEpsilon

```

```
    @minEpsilon.setter
    def minEpsilon(self, minEpsilon):
        self.__minEpsilon = minEpsilon

    @property
    def epsilonDecrease(self):
        return self.__epsilonDecrease

    @epsilonDecrease.setter
    def epsilonDecrease(self, epsilonDecrease):
        self.__epsilonDecrease = epsilonDecrease

    @property
    def goalReward(self):
        return self.__goalReward

    @goalReward.setter
    def goalReward(self, goalReward):
        self.__goalReward = goalReward

    @property
    def actionDictionary(self):
        return self.__actionDictionary

    @property
    def numOfEpisodes(self):
        return self.__numOfEpisodes

    def setEpsilonDecrease(self, epsilonDecrease):
        self.__epsilonDecrease = epsilonDecrease

    def setGoalReward(self, goalReward):
        self.__goalReward = goalReward

    def setConvergenceInterval(self, convergenceInterval):
        self.__convergenceInterval = int(convergenceInterval)

    @property
    def meanValues(self):
        return self.__meanValues

    @property
    def totalRewardArray(self):
        return self.__totalRewardArray

    def getAction(self, index):         # index 0-3 # actionDictionary 1-4
        for action, value in self.__actionDictionary.items():
            if value == index+1:
                return action

    def applyQLearningPS(self):

        stateList = self.__environment.stateList

        numOfActionsInEpisode = totalNumOfActions = 0
        self.__currentEpsilon = self.__maxEpsilon

        var = 1

        # episode loop
        while var != 0 and self.__currentEpsilon > self.__minEpsilon:

            agentCurrentCoordinates = self.__environment.startCoordinates  # initial coordinates of the
    agent,
            # startCoordinates is 2 dimensional

            self.__numOfEpisodes += 1
            numOfActionsInEpisode = 0

            # action loop
            currentState = stateList[agentCurrentCoordinates]  # current state in state number

            # 1 episode finishes when the agent reaches the goal state or the punishment state
            while not (currentState.isGoal or currentState.isPunishment):

                currentState = stateList[agentCurrentCoordinates]

```

```
            chosenAction = currentState.choseAction(self.__currentEpsilon)  # chose an action based on epsilon
            chosenActionIndex = self.__actionDictionary.get(chosenAction) - 1   # index 0-3

            # get the destination coordinates
            destinationCoordinates = self.__environment.getDestState(agentCurrentCoordinates, chosenAction)

            destinationState = stateList[destinationCoordinates]  # get the destination state

            self.__Model[currentState.stateNo][chosenActionIndex] = (
                destinationState.stateNo, destinationState.reward)


            p = abs(destinationState.reward + self.__gamma * destinationState.getMaxQValue() -
                    currentState.getQValues(chosenAction))

            if p > self.__theta:
                self.__PQueue.insert(p, (currentState.stateNo, chosenActionIndex))

            for i in range(self.__N):
                if not self.__PQueue.isEmpty():
                    currentStateNo, chosenActionIndex = self.__PQueue.delete()

                    currentState = self.__environment.getState(currentStateNo)

                    chosenAction = self.getAction(chosenActionIndex)

                    destinationStateNo, currentReward = self.__Model[currentStateNo][chosenActionIndex]
                    destinationState = self.__environment.getState(destinationStateNo)

                    # update the QValue
              #      print("updateQValue:", currentState.stateNo, chosenAction)
                    currentState.setQValues(chosenAction, currentState.getQValues(chosenAction) + self.__alpha * (
                            currentReward + self.__gamma * destinationState.getMaxQValue() -
                            currentState.getQValues(chosenAction)))

                    # trace model to find s'' and a''
                    for state_idx in range(self.__modelSize):
                        for action_idx in range(4):             # 4 for actions
                            if self.__Model[state_idx][action_idx][0] == currentState.stateNo:

                                tempCurrentState = self.__environment.getState(state_idx)
                                tempChosenAction = self.getAction(action_idx)

                                r_prev = self.__Model[state_idx][action_idx][1]
                                p = abs(r_prev + self.__gamma * currentState.getMaxQValue() -
                                        tempCurrentState.getQValues(tempChosenAction))
                                if p > self.__theta:
                                    self.__PQueue.insert(p, (tempCurrentState.stateNo, action_idx))

            agentCurrentCoordinates = destinationCoordinates
            numOfActionsInEpisode += 1

            currentState = stateList[agentCurrentCoordinates]       # next step

        # 1 episode finishes

        print("PS currentEpsilon:", self.__currentEpsilon)

        # convergence calculation

        # if the lastNSteps array is full, shift the array and get the new value
        if self.__numOfEpisodes > self.__convergenceInterval:
            self.__lastNSteps[0:self.__convergenceInterval - 1] = self.__lastNSteps[1:]
            self.__lastNSteps[self.__convergenceInterval - 1] = numOfActionsInEpisode

        # if the lastNSteps array is not full, get the values
        else:
            self.__lastNSteps[self.__numOfEpisodes - 1] = numOfActionsInEpisode

        totalNumOfActions += numOfActionsInEpisode

        # mean values of the number of actions in episodes calculation

        if len(self.__meanValues) == 0:
```

```
                self.__meanValues.append(numOfActionsInEpisode / self.__numOfEpisodes)
            else:
                lastMean = self.__meanValues[-1]
                self.__meanValues.append(
                    (lastMean * (self.__numOfEpisodes - 1) + numOfActionsInEpisode) / self.__numOfEpisodes)

            self.__currentEpsilon -= self.__epsilonDecrease
            var = variance(self.__lastNSteps)
            print("PS␣variance:", var)

        # print Qvalues
        self.printQValues(stateList, "PS")

    def printQValues(self, stateList, method):
        print(method)
        for i in range(0, self.__environmentSize[0]):
            for j in range(0, self.__environmentSize[1]):
                print(i, j)
                if 'L' in stateList[i][j].validActions:
                    print("L:", stateList[i][j].getQValues("L"), "␣")
                if 'R' in stateList[i][j].validActions:
                    print("R:", stateList[i][j].getQValues("R"), "␣")
                if 'U' in stateList[i][j].validActions:
                    print("U:", stateList[i][j].getQValues("U"), "␣")
                if 'D' in stateList[i][j].validActions:
                    print("D:", stateList[i][j].getQValues("D"), "␣")
```

Listing 4: Python - PS.py

### *E. Environment.py*

```
import numpy as np
from State import State


class Environment:

    def __init__(self, size):

        self.__environmentSize = size
        self.__stateList = np.empty(shape=self.__environmentSize, dtype=State)
        self.__goalReward = 500
        self.__punishment = -500
        self.__startCoordinates = (int(self.__environmentSize[0] / 2), int(self.__environmentSize[1] / 2))
        self.__goalCoordinates = (self.__environmentSize[0] - 1, self.__environmentSize[1] - 1)
        self.__punishmentCoordinates = (0, 0)

        stateNo = 0
        # generate states and put into stateList
        for row in range(0, self.__environmentSize[0]):
            for col in range(0, self.__environmentSize[1]):
                self.__stateList[row][col] = State(stateNo, self.__environmentSize)
                stateNo += 1

        # goal state
        self.__stateList[self.__environmentSize[0] - 1][self.__environmentSize[1] - 1].isGoal = True

        # punishment state
        self.__stateList[0][0].isPunishment = True

    @property
    def environmentSize(self):
        return self.__environmentSize

    @property
    def stateList(self):
        return self.__stateList

    @property
    def goalReward(self):
        return self.goalReward

    @goalReward.setter
    def goalReward(self, goalReward):
```

```
        self.__goalReward = goalReward

    @property
    def punishment(self):
        return self.__punishment

    @punishment.setter
    def punishment(self, punishment):
        self.__punishment = punishment

    @property
    def startCoordinates(self):
        return self.__startCoordinates

    @property
    def goalCooridinates(self):
        return self.__goalCoordinates

    @property
    def punishmentCoordinates(self):
        return self.__punishmentCoordinates

    def getState(self, index):
        for row in range(0, self.__environmentSize[0]):
            for col in range(0, self.__environmentSize[1]):
                if self.__stateList[row][col].stateNo == index:
                #    print("index:", index, "row:", row, "col:", col, "ee:", (self.__stateList[row][col]).stateNo)
                    return self.__stateList[row][col]

    def getDestState(self, coordinate, action):
        if action == ’L’:
            destCoordinate = coordinate[0], coordinate[1] - 1
        elif action == ’R’:
            destCoordinate = coordinate[0], coordinate[1] + 1
        elif action == ’U’:
            destCoordinate = coordinate[0] - 1, coordinate[1]
        elif action == ’D’:
            destCoordinate = coordinate[0] + 1, coordinate[1]

        return destCoordinate
```

Listing 5: Python - Environment.py

### *F. State.py*

```
import random
import operator


class State:

    def __init__(self, stateNo, size):

        # state parameters
        self.__validActions = ["L", "R", "U", "D"]
        self.__stateNo = stateNo
        self.__environmentSize = size
        self.__reward = 0
        self.__policies = {"mu": {}}
        self.__isGoal = False
        self.__isPunishment = False
        self.__QValues = {"L": 0.0, "R": 0.0, "U": 0.0, "D": 0.0}

        # generate initial action set
        self.generateActions("mu")

    @property
    def validActions(self):
        return self.__validActions

    @property
    def stateNo(self):
        return self.__stateNo
```

```python

    @property
    def environmentSize(self):
        return self.__environmentSize

    @property
    def reward(self):
        return self.__reward

    @reward.setter
    def reward(self, reward):
        self.__reward = reward

    @property
    def policies(self):
        return self.__policies

    @property
    def isGoal(self):
        return self.__isGoal

    @isGoal.setter
    def isGoal(self, isGoal):
        if isGoal:
            self.__reward = 500
            self.__isGoal = True
        else:
            self.__reward = 0
            self.__isGoal = False

    @property
    def isPunishment(self):
        return self.__isPunishment

    @isPunishment.setter
    def isPunishment(self, isPunishment):
        if isPunishment:
            self.__reward = -500
            self.__isPunishment = True
        else:
            self.__reward = 0
            self.__isPunishment = False

    def getQValues(self, action):
        return self.__QValues[action]

    def setQValues(self, action, QValue):
        self.__QValues[action] = QValue

    def generateActions(self, policy="mu"):
        if policy in self.__policies:
            if self.__stateNo % self.__environmentSize[1] == 0:  # column 0
                self.__validActions.remove("L")
                self.__QValues.pop("L")
            if self.__stateNo % self.__environmentSize[1] == (self.__environmentSize[1]-1):  # last column
                self.__validActions.remove("R")
                self.__QValues.pop("R")
            if self.__stateNo in range(0, self.__environmentSize[1]):  # row 0
                self.__validActions.remove("U")
                self.__QValues.pop("U")
            if self.__stateNo in range((self.__environmentSize[0] - 1) * self.__environmentSize[1],
                                       self.__environmentSize[0] * self.__environmentSize[1]):  # bottom row
                self.__validActions.remove("D")
                self.__QValues.pop("D")

    def choseAction(self, epsilon):
        rand = random.random()  # random number between [0,1)
        if rand <= epsilon:
            chosenAction = self.explorative()
        else:
            chosenAction = self.exploitative()
        return chosenAction

    def explorative(self):
        return random.choice(list(self.__QValues))

    def exploitative(self):
```

```
        return max(self.__QValues.items(), key=operator.itemgetter(1))[0]

    def getMaxQValue(self):
        return self.__QValues[max(self.__QValues.items(), key=operator.itemgetter(1))[0]]
```

Listing 6: Python - State.py

## REFERENCES

[1] Sutton, R. and Barto, A. (n.d.). Reinforcement learning: an introduction. 2nd ed. pp.1-4, pp.103-106, pp.137-140.
[2] https://towardsdatascience.com/the-complete-reinforcement-learning-dictionary-e16230b7d24e (Date of Access:07/01/2020)