COMP 473 / COMP 6731
Student Id: 40083939
Student Name: Apoorv Semwal
Mail-Id: a_semwal@encs.concordia.ca

Assignment 3: Reinforcement Learning using SARSA (State-Action-Reward-State-Action) on a Windy Grid World.

Before diving headfirst into the given problem scenario it appears logical to first discuss a few foundational concepts and the terminologies related to the problem.

Environment: A, 7 X 10 Grid World where some agent has to **Start from State S (3, 0)** and **reach a Goal State G (3, 7). *Note:** Indexing for rows and columns is assumed to start from 0. Rows are starting – Top to Bottom and columns from Left to Right.

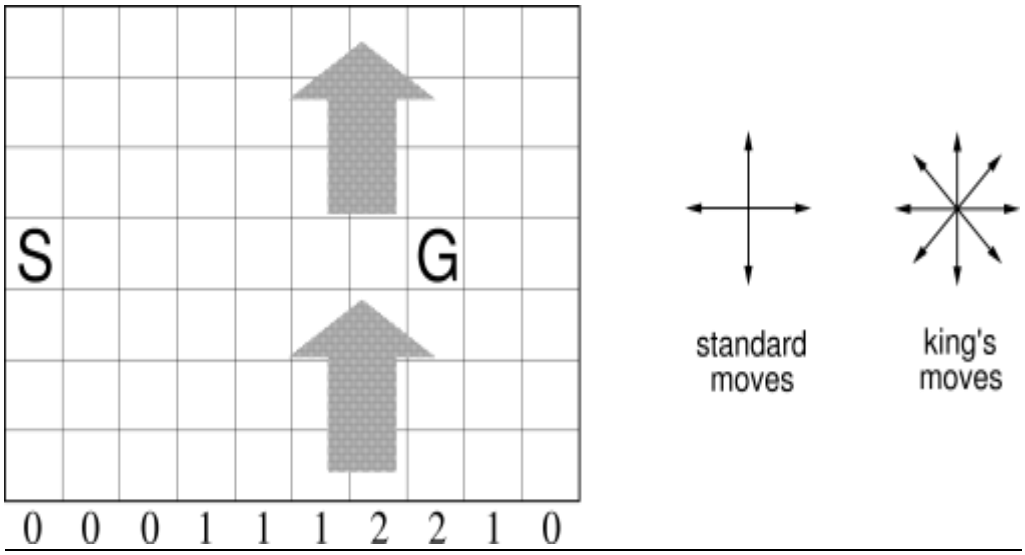
State: Every square in this 7 X 10 Grid World is considered to be a State where an agent can reach at some specific time step "ti".

Actions: In order to move from one state to another our agent needs to perform some specific actions. The 2 scenarios which have been given to us specifically list down the valid action moves.

a. **Standard Moves 4 – Actions** from any **state (squares in grid word)**, Move "**UP, DOWN, LEFT, RIGHT**"
b. **Kings Moves 8 – Actions** from any **state (squares in grid word)**, Move "**UP, DOWN, LEFT, RIGHT, UP-RIGHT, UP-LEFT, DOWN-RIGHT, DOWN-LEFT"**.

Wind Push: An additional constraint has been given in form wind blowing from the bottom of every column. This blow of wind has a specific intensity to blow our agent up either **0 or 1 or 2 rows UP from the actual row where the agent was supposed to end up**. How many rows the wind will blow our agent UP, is mentioned as the last row of this Grid World.

Pictorial representation of the elements discussed above.



Reward: A real valued number which our agent receives after performing some action on any particular state. **Reward value is the quantifiable measure to inform our agent how good or bad was its move of taking any particular action "a" on a particular state "S".**

Q-Values: It's the **utility value** or the **long term expected return/reward** of **taking an action "a"** at **time step "t"** while **agent was in state "S"**. **Taking action "a" on state "S"** will bring our agent to some **new State "S' "**, where our agent will again have to pick from a set of available actions and further move to a new state. **Every time our agent moves from S->S' taking an action "a", we update the Q-value of state S associated to action "a"**.

A more **functional representation** would treat **Q as a function** taking **one state-action pair as input** and giving the utility value or **long term expected reward** of **taking an action "a" in state "S"**, as output. **The term long term expected reward will be explained in the next point.**

Q(S,a)-> some real number (Utility of that state).

Temporal Differencing (TD) Learning: This approach of Reinforcement learning **overcomes** one of the **problems being faced in Monte Carlo (MC)** Simulation based Learning. Using MC our **agent needs to wait till the end of episode before it can actually update the utility function**. This was a serious problem as **some applications might have very long episodes resulting in delayed learning**. Moreover termination is not guaranteed at all in a couple of other scenarios. TD overcomes these problems.

In general any TD algorithm **predicts at every time-step**, the **total amount of reward expected over the future**, i.e. it is an estimation problem which, as stated by Andrew G. Barto, is:

"Suppose a system receives as input a time sequence of vectors (xt,yt) , t=0,1,2,… , where each xt is an arbitrary signal**(State of the system)** and yt**(Reward)** is a real number. TD learning applies to the problem of producing at each discrete time step t, an estimate, or prediction, of the following quantity:

$$Y_t = y_{t+1} + \gamma y_{t+2} + \gamma^2 y_{t+3} + \cdots = \sum_{i=1}^{\infty} \gamma^{i-1} y_{t+i},$$

Converting it to a recursive form where we can express it with just 3 Parameters:

Current Reward + (Gamma * Long Term Expected Reward from New State)

$$Y_t = y_{t+1} + \gamma y_{t+2} + \gamma^2 y_{t+3} + \cdots \ :$$

$$= y_{t+1} + \gamma[y_{t+2} + \gamma y_{t+3} + \gamma^2 y_{t+4} + \cdots] \ :$$

$$= y_{t+1} + \gamma Y_{t+1},$$

In context of Reinforcement learning $\mathbf{y_{t+i}}$ are the current and future rewards and at each time step and we try to predict the total sum. We say the word **prediction as at each time step we are considering future rewards as well**.

The **total sum Yt** here is actually the **long term expected reward** which was mentioned above.

Here Gamma (γ) is the discount factor with 0<= γ<=1. It encourages an agent to seek reward sooner than later. Basically represents **how much we care about getting rewards sooner than getting them later**. For instance Gamma = 1 means getting a reward at 1st time step is of same value as getting it in 1000th time step, whereas a Gamma value less than 1 and greater than 0 means getting a reward at 1st time step is of higher value than getting the same reward in 1000th time step, as the powers of Gamma keep increasing with time steps.

Specific Techniques of TD Learning:

The general rule behind any TD technique is:

$$\text{NewEstimate} \leftarrow \text{OldEstimate} + \text{StepSize}\big[\text{Target} - \text{OldEstimate}\big]$$

NewEstimate – of the Q Value for a given (State,Action) Pair.

OldEstimate – of the Q Value for a given (State,Action) Pair.

Target – Long term expected Rewards for selecting a particular action in any state.

StepSize – Alpha–Learning Rate –0<= Alpha <=1– Determines how much of the difference between the newly proposed Q-Value and the previous Q-Value should be taken into consideration. It lets us increase Q values in small steps over multiple episodes.

A more mathematical formulation for it is called as the **"The Bellman's Equation."**

$$V(S_t) \leftarrow V(S_t) + \alpha\Big[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\Big]$$

New Estimate | Old Estimate | Step Size | Target | Old Estimate

The two most Commonly Used Algorithms for TD Learning are:

a. SARSA
b. Q-Learning

Both these algorithms make use of the above mentioned Bellman's Equation with just a **difference in the way** they calculate **TARGET Factor in the above equation**.

For **SARSA** equation takes the form:

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

That is for calculating the new estimate SARSA **always considers the action which is actually being taken in the new state as part of current policy,** that is why it is classified as **ON-POLICY**.

For **Q-learning** equation takes the form

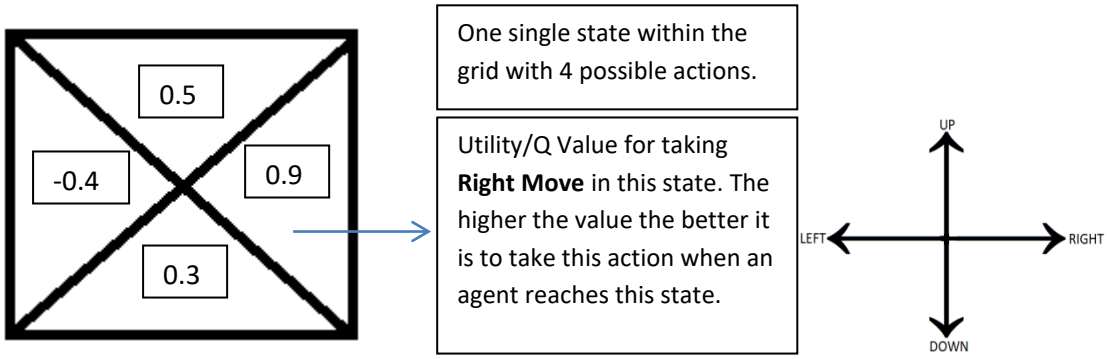
$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$

That is for calculating the new estimate Q-Learning **always considers the action with highest Q-value in the new state**, **irrespective of the policy being followed** that is why it is classified as **OFF-POLICY**.

==========

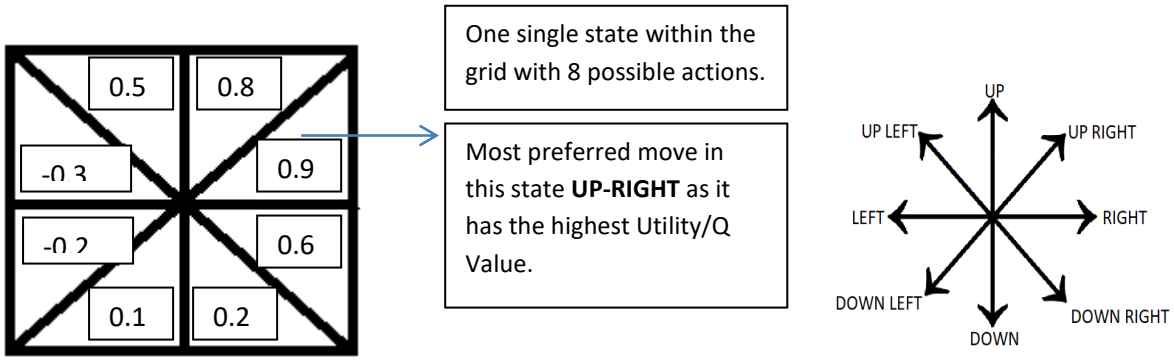
For the windy grid world scenario we have already been given the simulation results with 4 standard moves taken into consideration and are being asked to check if there will be any improvements with 8 moves possible (Kings Moves).

To analyse that first let us **model each state in the grid along with the actions possible and their Utility/Q values**.



So that way we can assume the entire **Environment as a (Rows- 7 * Cols-10 * Actions-4) Array**

Same way **for 8 actions (Kings Move) each state would look something like**:



So that way we can assume the entire **Environment as a (Rows- 7 * Cols-10 * Actions-8)3D - array**

***Note – below given are just the pseudo code snippets to get an understanding of how the actual working code might look like. (Some basic Python syntax has been used) By no means should it be treated as an actual code.**

Start by Initialising Environment (Python Syntax):

```
import numpy as np
ROWS_GRID   = 7
COLS_GRID   = 10
environment = np.zeros((ROWS_GRID, COLS_GRID, 8)) #Since we have 8 actions
#This will set all utility/Q Values to 0 for every action in the state.

max_episode_count = 600
#This will set a limit on maximum number of episodes (1 episode is one full
#run of our agent from Start to Goal State).
```

Since we have been asked to **use SARSA to arrive at an optimal Policy with 8 moves** so we will **update the utility values based on SARSA.**

Pseudo Code as given by Richard S. Sutton and Andrew G. Barto in their book "Reinforcement Learning: An Introduction"

Loop for each episode:
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma Q(S',A') - Q(S,A)\right]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

```
#Now we will be defining our start and goal state, possible moves, reward for
#each move, Epsilon and the learning rate

# START and GOAL STATE
START_STATE = [3, 0]
GOAL_STATE = [3, 7]

# Possible moves
ACTION_UP = 0
ACTION_DOWN = 1
ACTION_LEFT = 2
ACTION_RIGHT = 3
ACTION_UP_RIGHT = 4
ACTION_UP_LEFT = 5
ACTION_DOWN_RIGHT = 6
ACTION_DOWN_LEFT = 7
```

```
# Complete list of actions
ACTIONS = [ACTION_UP, ACTION_DOWN, ACTION_LEFT, ACTION_RIGHT, ACTION_UP_RIGHT,
ACTION_UP_LEFT, ACTION_DOWN_RIGHT, ACTION_DOWN_LEFT]

# List representing wind for each column in the same sequence from Left to
Right
WIND_PUSH = [0, 0, 0, 1, 1, 1, 2, 2, 1, 0]

# Function actually implementing the move – taking agent from one state to
# another
def create_move(current_state, current_action):
    i, j = current_state
#State position is defined by a row - i and column - j
    if current_action == ACTION_UP:
        return [max(i - 1 – WIND_PUSH[j], 0), j]
    elif current_action == ACTION_DOWN:
        return [max(min(i + 1 - WIND_PUSH[j], ROWS_GRID - 1), 0), j]
    elif current_action == ACTION_LEFT:
        return [max(i - WIND_PUSH[j], 0), max(j - 1, 0)]
    elif current_action == ACTION_RIGHT:
        return [max(i - WIND_PUSH[j], 0), min(j + 1, COLS_GRID - 1)]
    elif current_action == ACTION_UP_RIGHT:
        return [max(i - WIND_PUSH[j] - 1, 0), min(j + 1, COLS_GRID - 1)]
    elif current_action == ACTION_UP_LEFT:
        return [max(i - WIND_PUSH[j] - 1, 0), max(j - 1, 0)]
    elif current_action == ACTION_DOWN_RIGHT:
        return [max(min(i + 1 - WIND_PUSH[j], ROWS_GRID - 1), 0), min(j +
                1, COLS_GRID - 1)]
    elif current_action == ACTION_DOWN_LEFT:
        return [max(min(i + 1 - WIND_PUSH[j], ROWS_GRID - 1), 0), max(j -
                1, 0)]

# Probability for exploration
EPSILON = 0.1

# Learning Rate
ALPHA = 0.6

# Discount Factor
GAMMA = 0.7

# Reward for each move
REWARD = -1.0
```

```
episode_count = 0
while episode_count < max_episode_count:

    # Starting the episode from start state defined above
    current_state = START_STATE

    # choose an action based on epsilon greedy
    # Generate a random number between 0 and 1.
    if random_number <= EPSILON:
        current_action = #Exploration – Choose a Random action from above
                         #defined actions
    else:
        current_action = #Exploitation-based on current_state maximum Q
                        #value across all possible actions in that state

    # moving the agent until it reaches the goal state

    while current_state != GOAL_STATE:

      next_state = create_move(current_state, current_action) #Calling create
                                                              #move defined above

      #Now since this is SARSA we have to run epsilon greedy for the next
      #state as well and get the action
      if random_number <= EPSILON:
            next_action = #Exploration – Choose a Random action from above
                          #defined actions
      else:
           next_action = #Exploitation-based on next_state maximum Q
                         #value across all possible actions in that state

      #Now we do the SARSA Update as we have all the required parameters for
      #it – current_state, current_action, next_state, next_action, reward
      #learning_rate, discount_rate.

                  Value(current_state,current_action) =
                  Value(current_state,current_action) +
        ALPHA * (REWARD + (GAMMA * (Value(next_state,next_action))) -
                  Value(current_state,current_action))

      #Again the same cycle starts for the next_state as the new
      #current_state and next_action as the current_action.

      current_state = next_state

      current_action = next_action

#BOTH INNER (To Reach GOAL_STATE) AND OUTER (To Run New Episodes) LOOP END
HERE.

*=========================================================================*
```

SOME SAMPLE ITERATIONS

Initially at the beginning of first episode all the Q values will be ZERO and over a period of time, after certain iterations let us assume we achieve the below given states.

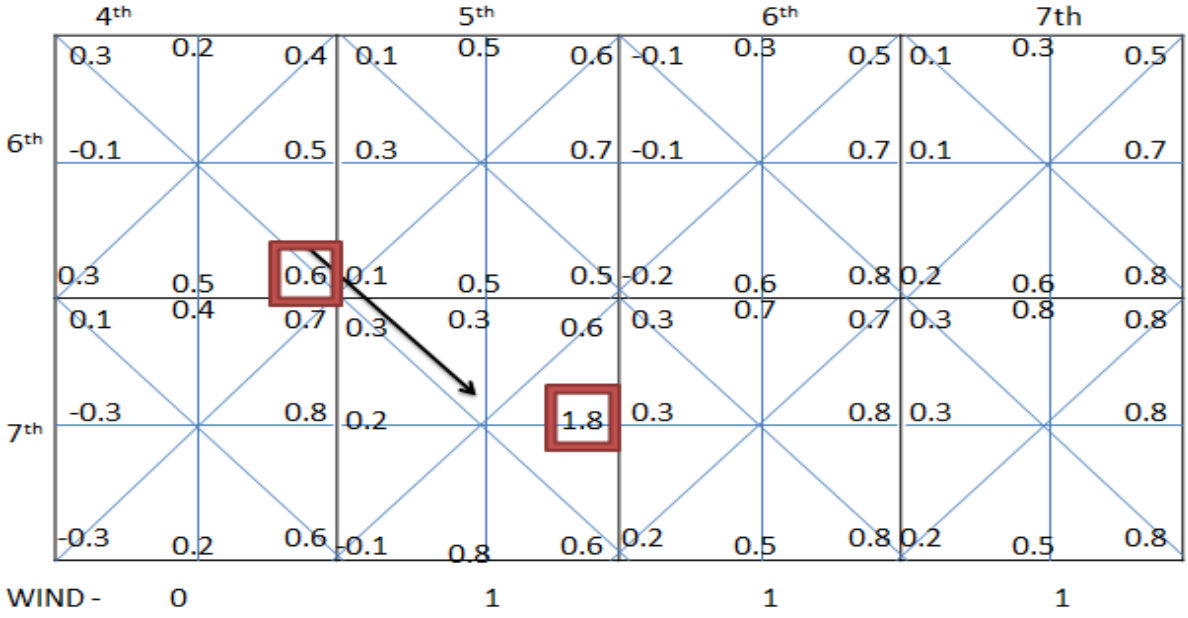
Learning Rate Alpha - is assumed to be 0.5

Discount Rate Gamma - is assumed to be 0.9

At each step Reward = -1

For the purpose of understanding we have taken 8 states out of 70 available and will observe how the utility values will update in successive iterations.

- We have take 4th, 5th, 6th, 7th Columns and 6th , 7th row.
- Agent is in 6th Row and 4th Column.
- Current Utility/Q Values have been shown for every action in each state.



Iteration 1 from this point:

Agent was in State(6,4) 6th Row 4th Col. Based on Epsilon Greedy Action Obtained was DOWN_RIGHT as it had the maximum value 0.6

This move brings us to State(7,5). Again based on Epsilon Greedy. Action Obtained was RIGHT as it had the maximum value 1.8

So to update Value for Original State(6,4) we use SARSA Update

NewValState(6,4,DOWN_RIGHT) =

CurrentStateOldVal(6,4,DOWN_RIGHT) + Alpha(Reward + (Gamma * (NextStateVal (7,5, RIGHT))) - CurrentStateOldVal (6,4,DOWN_RIGHT))

= 0.6 + 0.5 (-1 + (0.9 * 1.8) - 0.6) = 0.6 + 0.5 (0.02) = 0.6 + 0.01 = 0.61

NewValState(6,4,DOWN_RIGHT) = **0.61**

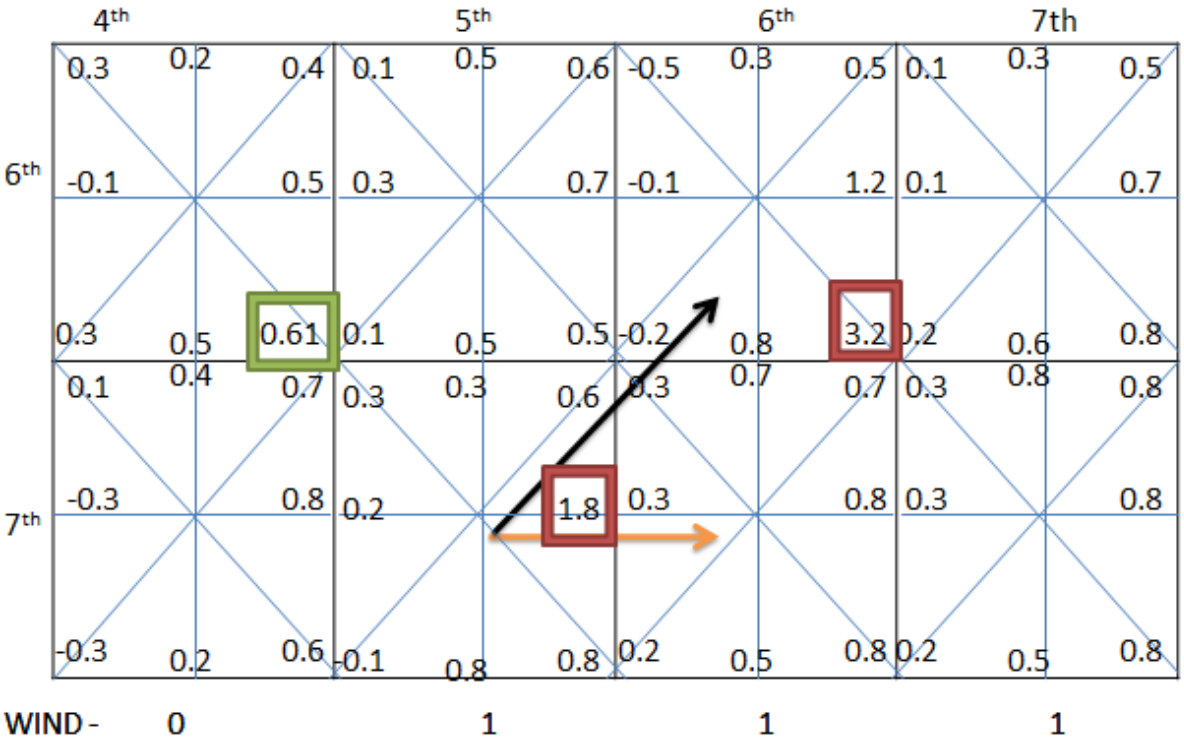
CurrentState = NextState –> Next State becomes the CurrentState for Next Iteration

Iteration 2:

Green Square shows the newly Updated Value for State(6,4,DOWN_RIGHT) = 0.61

Now the CurrentState is (7,5) and action chosen is RIGHT with Q-Value = 1.8.

Even though Agent tries to move RIGHT but Wind Intensity from Column pushes it up by one Sqaure and the NextState is (6,6) with action DOWN_RIGHT in it obtained from Epsilon Greedy Policy.



NewValState(7,5, RIGHT) =

CurrentStateOldVal(7,5, RIGHT) + Alpha(Reward + (Gamma * (NextStateVal (6,6,DOWN_ RIGHT))) - CurrentStateOldVal(7,5, RIGHT))

= 1.8 + 0.5 (-1 + (0.9 * 3.2) – 1.8) = 0.6 + 0.5 (0.08) = 1.8 + 0.04 = 1.84

NewValState(7,5, RIGHT) = **1.84**

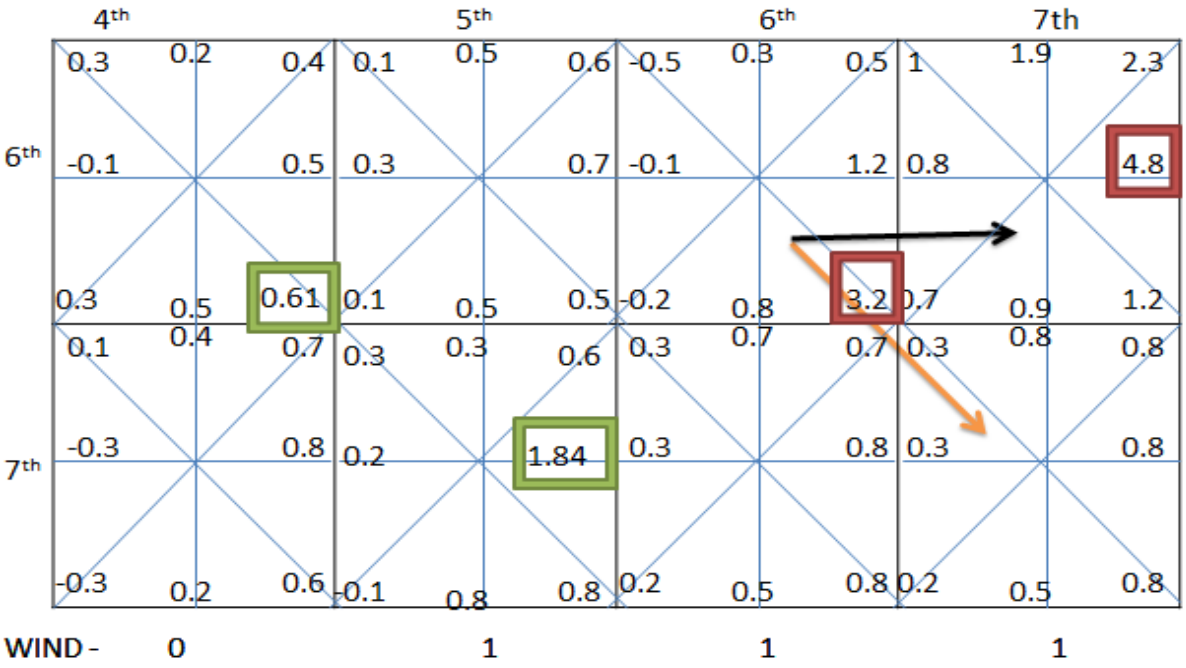
CurrentState = NextState –> Next State becomes the CurrentState for Next Iteration

Iteration 3:

Green Square shows the newly Updated Value for State(7,5, RIGHT) = 1.84

Now the CurrentState is (6,6) and action chosen is DOWN_RIGHT with Q-Value = 3.2.

Even though Agent tries to move DOWN_RIGHT but Wind Intensity from Column pushes it up by one Sqaure and the NextState is (6,7) with action UP_RIGHT in it obtained from Epsilon Greedy Policy.



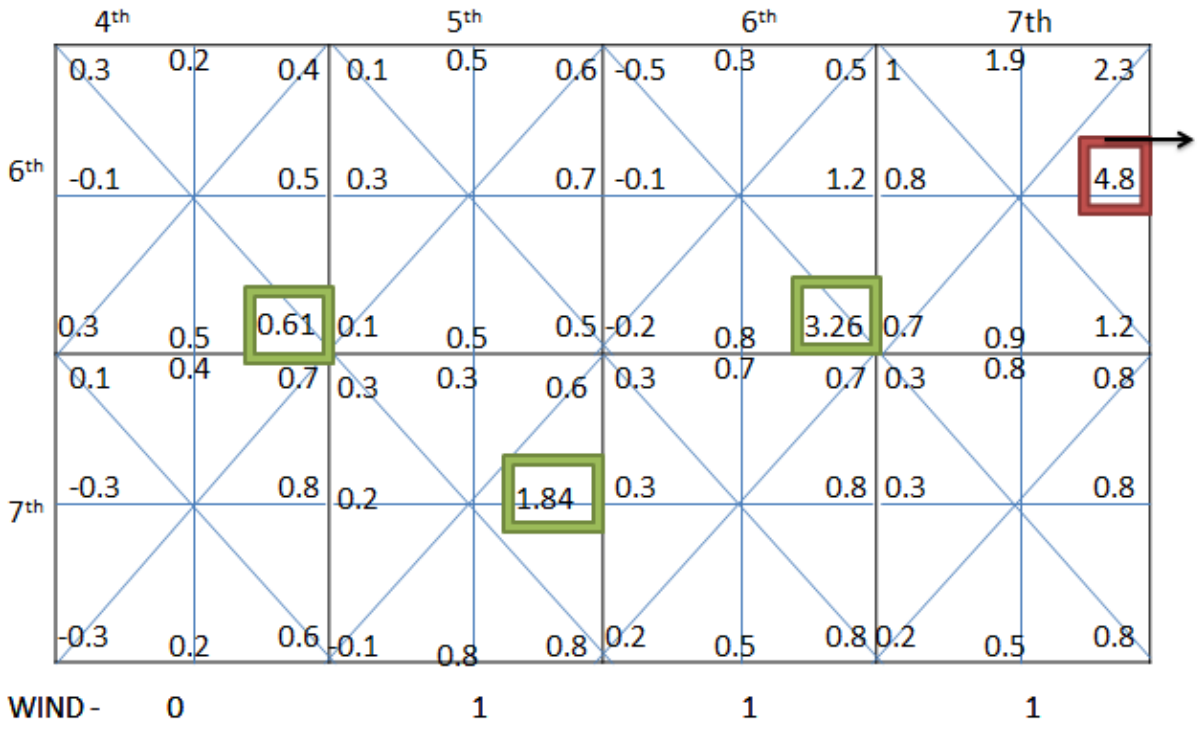
NewValState(6,6, DOWN_RIGHT) =

CurrentStateOldVal(6,6, DOWN_RIGHT) + Alpha(Reward + (Gamma * (NextStateVal (6,7, RIGHT))) - CurrentStateOldVal(6,6, DOWN_RIGHT))

= 3.2 + 0.5 (-1 + (0.9 * 4.8) – 3.2) = 0.6 + 0.5 (0.12) = 3.2 + 0.06 = 3.26

NewValState(6,6, DOWN_RIGHT) = **3.26**

CurrentState = NextState –> Next State becomes the CurrentState for Next Iteration



Likewise steps will continue until an episode ends in a Goal State.

Once an episode ends a new episode will start with updated values and we hope to achieve an optimal policy at the end of maximum number of episodes set.

OPTIMAL POLICY WITH KINGS MOVEs

Notation used:

S- Start State, G-Goal State

Action – R – Right, DR – Down Right

As we can observe at

Step – 4: Agent took R- Right but wind pushed it 1 square UP

(ORANGE ARROW – INTENDED MOVE)

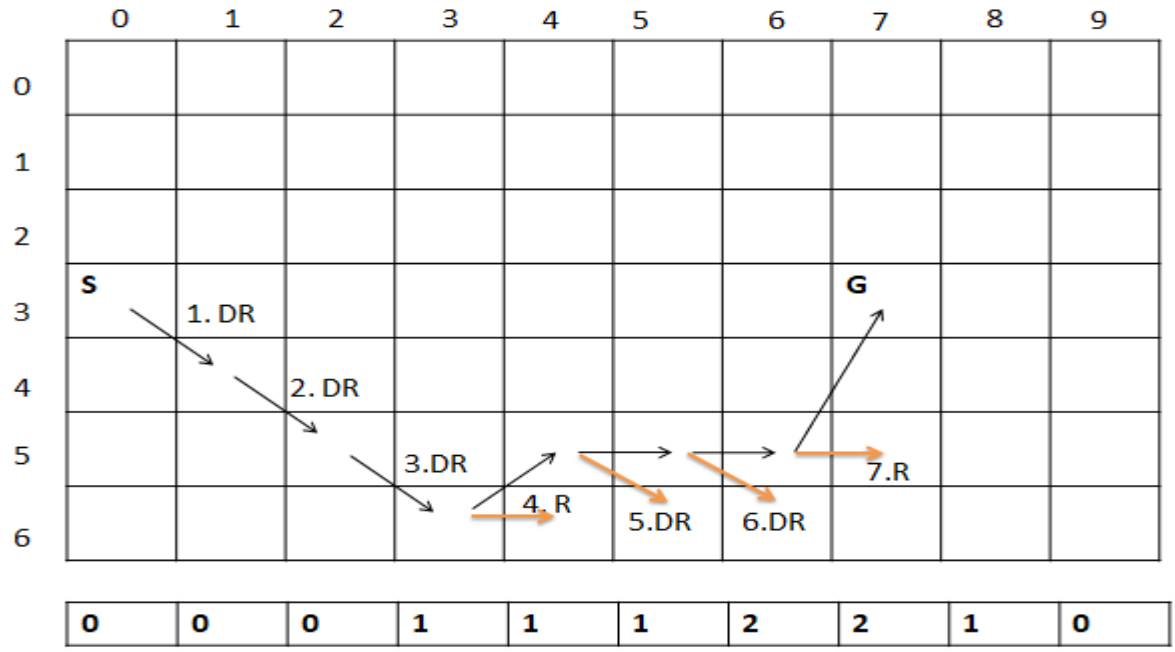
(BLACK ARROW – ACTUAL MOVE CONSIDERING PUSH FROM WIND)

Step – 5: Agent took DR- Down Right but wind pushed it 1 square UP

Step – 6: Agent took DR- Down Right but wind pushed it 1 square UP

Step – 7: Agent took R - Right but wind pushed it 2 squares UP

GOAL STATE ACHIEVED IN JUST – 7 steps with 8 possible actions, which is lesser than what we could achieve optimally with 4 possible actions. With 4 possible actions our optimal Policy had 15 steps.



WIND SPEED FOR EACH COLUMN

Adding a NEW 9th – **NO_ACTION** to the existing action SET where we only consider the push by wind **cannot help us to achieve a better Optimal Policy which is less than 7 time Steps**. Reason being:

The Goal state G and Start State S are separated by 7 time steps which is the least we need to reach goal from the start state.

This can be proved by changing the above algorithm and adding a New NO_ACTION in the action set.

Run multiple simulations after this change.

Change can be done as:

```
# Possible moves
ACTION_UP = 0
ACTION_DOWN = 1
ACTION_LEFT = 2
ACTION_RIGHT = 3
ACTION_UP_RIGHT = 4
ACTION_UP_LEFT = 5
ACTION_DOWN_RIGHT = 6
ACTION_DOWN_LEFT = 7
ACTION_NO        = 8

# Complete list of actions
ACTIONS = [ACTION_UP, ACTION_DOWN, ACTION_LEFT, ACTION_RIGHT, ACTION_UP_RIGHT,
ACTION_UP_LEFT, ACTION_DOWN_RIGHT, ACTION_DOWN_LEFT, ACTION_NO

# List representing wind for each column in the same sequence from Left to
Right
```

```
WIND_PUSH = [0, 0, 0, 1, 1, 1, 2, 2, 1, 0]

# Function actually implementing the move – taking agent from one state to
# another
def create_move(current_state, current_action):
    i, j = current_state
#State position is defined by a row - i and column - j
    if current_action == ACTION_UP:
        return [max(i - 1 – WIND_PUSH[j], 0), j]
    elif current_action == ACTION_DOWN:
        return [max(min(i + 1 - WIND_PUSH[j], ROWS_GRID - 1), 0), j]
    elif current_action == ACTION_LEFT:
        return [max(i - WIND_PUSH[j], 0), max(j - 1, 0)]
    elif current_action == ACTION_RIGHT:
        return [max(i - WIND_PUSH[j], 0), min(j + 1, COLS_GRID - 1)]
    elif current_action == ACTION_UP_RIGHT:
        return [max(i - WIND_PUSH[j] - 1, 0), min(j + 1, COLS_GRID - 1)]
    elif current_action == ACTION_UP_LEFT:
        return [max(i - WIND_PUSH[j] - 1, 0), max(j - 1, 0)]
    elif current_action == ACTION_DOWN_RIGHT:
        return [max(min(i + 1 - WIND_PUSH[j], ROWS_GRID - 1), 0), min(j +
                1, COLS_GRID - 1)]
    elif current_action == ACTION_DOWN_LEFT:
        return [max(min(i + 1 - WIND_PUSH[j], ROWS_GRID - 1), 0), max(j -
                1, 0)]
```

```
elif action == ACTION_NO:

    return [max(i - WIND[j], 0), j]
```

Time taken by a single episode to end:

With 4- standard moves < With 8-Kings moves < With 9- moves

So overall its best to run this problem with 8-Kings moves as it helps us in achieving a policy with a minimum of 7 time steps.

REFERENCES :

Reinforcement Learning: An Introduction By Richard S. Sutton and Andrew G. Barto

Second Edition MIT Press, Cambridge, MA, 2017

http://www.scholarpedia.org/article/Temporal_difference_learning

https://mpatacchiola.github.io/blog/2017/01/29/dissecting-reinforcement-learning-3.html