Assignment 2 - Expected SARSA (Group 13)

Name	Student ID
Dhanush Kovi	CB.EN.U4AIE21025
Aakash Jammula	CB.EN.U4AIE21019
Harshanth CS	CB.EN.U4AIE21016
Govind R Nair	CB.EN.U4AIE21013

This notebook contains a Python implementation for the Expected SARSA algorithm for maze environment.

Reference: (given in question)

https://github.com/DavidMouse1118/Reinforcement-Learning-Maze-World/blob/master/RL_brain_expected_sarsa.py

Modifications: Added energy constraint to the maze environment. 1. The agent has a limited amount of energy. The agent can move in the maze until the energy is exhausted. The agent will be reset to the start position if the energy is exhausted. 2. If the agent has reached the goal state and the energy is not exhausted, then the reward is increased by remaining energy. 3. The goal is to reach the goal state with maximum energy remaining.

GitHub Repository: https://github.com/dkvc/SpringSem_RLAssg2

Packages

```
import os
import pickle
import time

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import tkinter as tk
```

Parameters

```
# Debug flag
DEBUG = 1

# Maze parameters
MAZE_H = 10
MAZE_W = 10
UNIT = 40

# Simulation parameters
sim_speed = 0.05
showRender = False
```

```
episodes = 10000
renderEveryNth = 10000
printEveryNth = 100
do_plot_rewards = True
```

```
# Agent & Environment parameters
agentXY = [0, 0]
goalXY = [4, 4]

wall_shape=np.
 ↪array([[7,4],[7,3],[6,3],[6,2],[5,2],[4,2],[3,2],[3,3],[3,4],[3,5],[3,6],[4,6],[5,6]])
pits=np.array([[1,3],[0,5], [7,7]])
```

Expected SARSA Algorithm

The implementation of Expected SARSA algorithm is same as given in the reference.

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

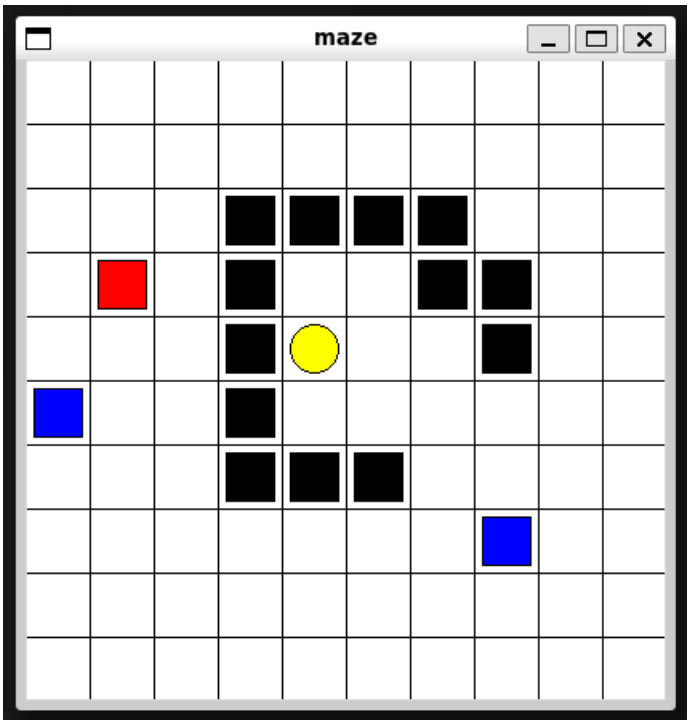
Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

```
class rlalgorithm:

    def __init__(self, actions, learning_rate=0.01, reward_decay=0.9,␣
 ↪e_greedy=0.1):
        self.actions = actions
        self.lr = learning_rate
        self.gamma = reward_decay
        self.epsilon = e_greedy
        self.q_table = pd.DataFrame(columns=self.actions, dtype=np.float64)
        self.display_name="Expected Sarsa"
        print(f"Using {self.display_name} ...")

    '''Choose the next action to take given the observed state using an epsilon␣
 ↪greedy policy'''
    def choose_action(self, observation):
```

```
        # Add non-existing state to our q_table
        self.check_state_exist(observation)

        # Select next action
        if np.random.uniform() >= self.epsilon:
            # Choose argmax action
            state_action_values = self.q_table.loc[observation, :]
            action = np.random.choice(state_action_values[state_action_values
==== np.max(state_action_values)].index) # handle multiple argmax with random
        else:
            # Choose random action
            action = np.random.choice(self.actions)

        return action


    '''Update the Q(S,A) state-action value table using the latest experience
       This is a not a very good learning update
    '''
    def learn(self, s, a, r, s_):
        self.check_state_exist(s_)
        q_current = self.q_table.loc[s, a]

        if s_ != 'terminal':
            # calculate expected value according to epsilon greedy policy
            state_action_values = self.q_table.loc[s_,:]
            value_sum = np.sum(state_action_values)
            max_value = np.max(state_action_values)
            max_count = len(state_action_values[state_action_values ==
max_value])
            k = len(self.actions) # total number of actions

            expected_value_for_max = max_value * ((1 - self.epsilon) /
max_count + self.epsilon / k) * max_count
            expected_value_for_non_max = (value_sum - max_value * max_count) *
(self.epsilon / k)

            expected_value = expected_value_for_max + expected_value_for_non_max

            q_target = r + self.gamma * expected_value # max state-action value
        else:
            q_target = r  # next state is terminal

        self.q_table.loc[s, a] += self.lr * (q_target - q_current)  # update
current state-action value

        return s_, self.choose_action(str(s_))
```

```
    '''States are dynamically added to the Q(S,A) table as they are␣
↪encountered'''
    def check_state_exist(self, state):
        if state not in self.q_table.index:
            self.q_table = pd.concat([self.q_table, pd.DataFrame([[0]*len(self.
↪actions)], columns=self.q_table.columns, index=[state])])

    def save(self, filename):
        os.makedirs(os.path.dirname(filename), exist_ok=True)
        with open(filename, 'wb') as file:
            pickle.dump(self.q_table, file)

    def load(self, filename):
        with open(filename, 'rb') as file:
            self.q_table = pickle.load(file)
```

Building Maze Environment



The maze environment consists of `MAZE_H` x `MAZE_W` grid. The agent can move in four directions: up, down, left, and right. 1. The agent receives a reward of -0.1 for each step taken in the maze. 2. The agent receives a reward of 1 if it reaches the goal state. 3. The agent receives a reward of -0.3 if it reaches the wall. 4. The agent receives a reward of -10 if it falls into pit. 4. The agent receives a reward of -20 if energy is exhausted.

```
origin = np.array([UNIT/2, UNIT/2])

class Maze(tk.Tk, object):
    def __init__(self, agentXY, goalXY, walls=[],pits=[], energy_capacity=50,␣
 ↪energy_factor=0.01):
        super(Maze, self).__init__()
        self.action_space = ['u', 'd', 'l', 'r']
        self.n_actions = len(self.action_space)
        self.wallblocks = []
        self.pitblocks=[]
        self.UNIT = UNIT
        self.MAZE_H = MAZE_H
        self.MAZE_W = MAZE_W
        self.energy_capacity = energy_capacity
        self.energy_factor = energy_factor
        self.energy = energy_capacity
        self.max_energy = 0
        self.title('maze')
        self.geometry('{0}x{1}'.format(MAZE_H * UNIT, MAZE_W * UNIT))
        self.build_shape_maze(agentXY, goalXY, walls, pits)
        #self.build_maze()

    def build_shape_maze(self,agentXY,goalXY, walls,pits):
        self.canvas = tk.Canvas(self, bg='white',
                                height=MAZE_H * UNIT,
                                width=MAZE_W * UNIT)

        # create grids
        for c in range(0, MAZE_W * UNIT, UNIT):
            x0, y0, x1, y1 = c, 0, c, MAZE_H * UNIT
            self.canvas.create_line(x0, y0, x1, y1)
        for r in range(0, MAZE_H * UNIT, UNIT):
            x0, y0, x1, y1 = 0, r, MAZE_W * UNIT, r
            self.canvas.create_line(x0, y0, x1, y1)


        for x,y in walls:
            self.add_wall(x,y)
        for x,y in pits:
            self.add_pit(x,y)
        self.add_goal(goalXY[0],goalXY[1])
        self.add_agent(agentXY[0],agentXY[1])
        self.canvas.pack()

    '''Add a solid wall block at coordinate for centre of bloc'''
    def add_wall(self, x, y):
        wall_center = origin + np.array([UNIT * x, UNIT*y])
```

```
        self.wallblocks.append(self.canvas.create_rectangle(
            wall_center[0] - 15, wall_center[1] - 15,
            wall_center[0] + 15, wall_center[1] + 15,
            fill='black'))

    '''Add a solid pit block at coordinate for centre of bloc'''
    def add_pit(self, x, y):
        pit_center = origin + np.array([UNIT * x, UNIT*y])
        self.pitblocks.append(self.canvas.create_rectangle(
            pit_center[0] - 15, pit_center[1] - 15,
            pit_center[0] + 15, pit_center[1] + 15,
            fill='blue'))

    '''Add a solid goal for goal at coordinate for centre of bloc'''
    def add_goal(self, x=4, y=4):
        goal_center = origin + np.array([UNIT * x, UNIT*y])

        self.goal = self.canvas.create_oval(
            goal_center[0] - 15, goal_center[1] - 15,
            goal_center[0] + 15, goal_center[1] + 15,
            fill='yellow')

    '''Add a solid wall red block for agent at coordinate for centre of bloc'''
    def add_agent(self, x=0, y=0):
        agent_center = origin + np.array([UNIT * x, UNIT*y])

        self.agent = self.canvas.create_rectangle(
            agent_center[0] - 15, agent_center[1] - 15,
            agent_center[0] + 15, agent_center[1] + 15,
            fill='red')

    def reset(self, value = 1, resetAgent=True):
        self.update()
        time.sleep(0.2)
        if(value == 0):
            return self.canvas.coords(self.agent)
        else:
            #Reset Agent
            if(resetAgent):
                self.canvas.delete(self.agent)
                self.agent = self.canvas.create_rectangle(origin[0] - 15,␣
↪origin[1] - 15,
                origin[0] + 15, origin[1] + 15,
                fill='red')
                self.energy = self.energy_capacity

            return self.canvas.coords(self.agent)
```

```
    '''computeReward - definition of reward function'''
    def computeReward(self, currstate, action, nextstate):
            reverse=False
            if self.energy < 0:
                reward = -20
                done = True
                nextstate = 'terminal'
                reverse = True
            elif nextstate == self.canvas.coords(self.goal):
                reward = 1
                done = True
                nextstate = 'terminal'
                self.max_energy = max(self.max_energy, self.energy)
            elif nextstate in [self.canvas.coords(w) for w in self.wallblocks]:
                reward = -0.3
                done = False
                nextstate = currstate
                reverse=True
            elif nextstate in [self.canvas.coords(w) for w in self.pitblocks]:
                reward = -10
                done = True
                nextstate = 'terminal'
                reverse=False
            else:
                reward = -0.1
                done = False

            if not done:
                reward += self.energy_factor * (self.energy_capacity - self.
↪energy)
            return reward,done, reverse

    '''step - definition of one-step dynamics function'''
    def step(self, action):
        s = self.canvas.coords(self.agent)
        base_action = np.array([0, 0])
        if action == 0:   # up
            if s[1] > UNIT:
                base_action[1] -= UNIT
        elif action == 1:   # down
            if s[1] < (MAZE_H - 1) * UNIT:
                base_action[1] += UNIT
        elif action == 2:   # right
            if s[0] < (MAZE_W - 1) * UNIT:
                base_action[0] += UNIT
        elif action == 3:   # left
```

```
            if s[0] > UNIT:
                base_action[0] -= UNIT

        self.canvas.move(self.agent, base_action[0], base_action[1])  # move
↪agent

        s_ = self.canvas.coords(self.agent)  # next state
        #print("s_.coords:{}({})".format(self.canvas.coords(self.
↪agent),type(self.canvas.coords(self.agent))))
        #print("s_:{}({})".format(s_, type(s_)))

        # call the reward function
        self.energy -= 1
        reward, done, reverse = self.computeReward(s, action, s_)
        if(reverse):
            self.canvas.move(self.agent, -base_action[0], -base_action[1])  #
↪move agent back
            s_ = self.canvas.coords(self.agent)

        #print(f"Energy: {self.energy}, Reward: {reward}, Done: {done}")
        return s_, reward, done

    def render(self, sim_speed=.01):
        time.sleep(sim_speed)
        self.update()

    def run_without_learning(self, agent, episodes=10, sim_speed=0.01):
        energies = []
        for _ in range(episodes):
            s = self.reset()
            while True:
                self.render(sim_speed)
                a = agent.choose_action(str(s))
                s_, r, done = self.step(a)
                if done:
                    energies.append(self.max_energy)
                    print(f"Energy: {self.max_energy}, Reward: {r}")
                    break
                s = s_
        print(f"Max Energy: {max(energies)}")
        self.destroy()
```

Running the Algorithm (Main Function)

Helper Functions

1. Debug Function

This function handles the debug prints in the code. It can be enabled by setting the `DEBUG` flag to `True`.

```
def debug(debuglevel, msg, **kwargs):
    if debuglevel <= DEBUG:
        if 'printNow' in kwargs:
            if kwargs['printNow']:
                print(msg)
        else:
            print(msg)
```

2. Plotting Function

   This function is used to plot the episode vs reward graph.

```
def plot_rewards(experiments):
    color_list=['blue','green','red','black','magenta']
    label_list=[]
    for i, (env, RL, data) in enumerate(experiments):
        x_values=range(len(data['global_reward']))
        label_list.append(RL.display_name)
        y_values=data['global_reward']
        plt.plot(x_values, y_values, c=color_list[i],label=label_list[-1])
        plt.legend(label_list)
    plt.title("Reward Progress", fontsize=24)
    plt.xlabel("Episode", fontsize=18)
    plt.ylabel("Return", fontsize=18)
    plt.tick_params(axis='both', which='major',
                    labelsize=14)
    plt.show()
```

3. Update Function

   This function is used to update the environment and the agent.

```
def update(env, RL, data, episodes=50):
    global_reward = np.zeros(episodes)
    data['global_reward']=global_reward

    for episode in range(episodes):
        t=0
        # initial state
        if episode == 0:
            state = env.reset(value = 0)
        else:
            state = env.reset()

        debug(2,'state(ep:{},t:{})={}'.format(episode, t, state))
```

```
        # RL choose action based on state
        action = RL.choose_action(str(state))
        while True:
            # fresh env
            #if(t<5000 and (showRender or (episode % renderEveryNth)==0)):
            if(showRender or (episode % renderEveryNth)==0):
                env.render(sim_speed)


            # RL take action and get next state and reward
            state_, reward, done = env.step(action)
            global_reward[episode] += reward
            debug(2,'state(ep:{},t:{})={}'.format(episode, t, state))
            debug(2,'reward_{}=  total return_t ={} Mean50={}'.format(reward,␣
↪global_reward[episode],np.mean(global_reward[-50:])))


            # RL learn from this transition
            # and determine next state and action
            state, action =  RL.learn(str(state), action, reward, str(state_))


            # break while loop when end of this episode
            if done:
                break
            else:
                t=t+1

        debug(1,"({}) Episode {}: Length={}  Total return = {} Energy Left =␣
↪{}".format(RL.display_name,episode, t,  global_reward[episode], env.
↪max_energy),printNow=(episode%printEveryNth==0))
        if(episode>=100):
            debug(1,"    Median100={} Variance100={}".format(np.
↪median(global_reward[episode-100:episode]),np.var(global_reward[episode-100:
↪episode])),printNow=(episode%printEveryNth==0))
    # end of game
    print('game over -- Algorithm {} completed'.format(RL.display_name))
    env.destroy()
```

Main Function

```
experiments = []

env = Maze(agentXY, goalXY, walls=wall_shape, pits=pits, energy_capacity=50,␣
 ↪energy_factor=0.01)
RL = rlalgorithm(actions=list(range(env.n_actions)))
```

```
data = {}

env.after(10, update(env, RL, data, episodes))
env.mainloop()
experiments.append((env, RL, data))

for env, RL, data in experiments:
    print(f"""
            [{RL.display_name}]: Max Reward = {np.max(data['global_reward'])},␣
 ↪MedLast100 = {np.median(data['global_reward'][-100:])}, VarLast100 = {np.
 ↪var(data['global_reward'][-100:])}
         """)

if do_plot_rewards:
    plot_rewards(experiments)
```

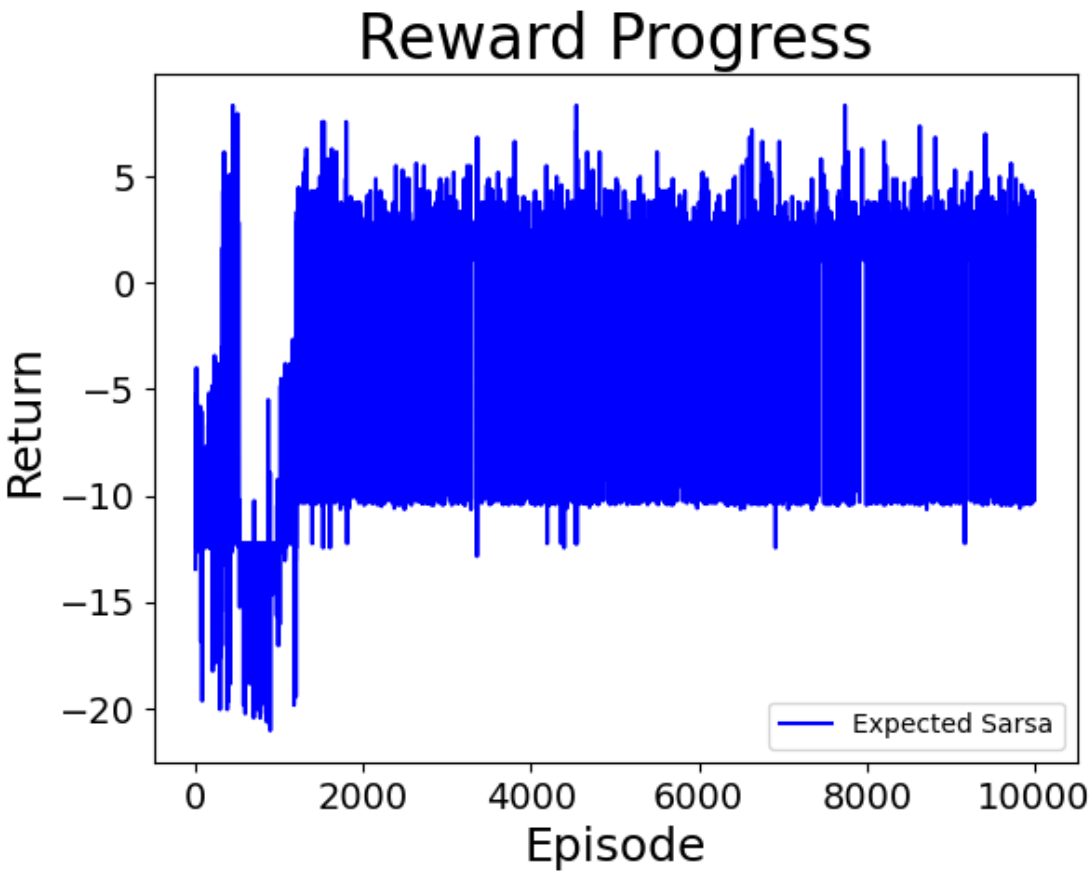
```
Using Expected Sarsa …
(Expected Sarsa) Episode 0: Length=50  Total return = -13.45 Energy Left = 0
(Expected Sarsa) Episode 100: Length=50  Total return = -12.45 Energy Left = 0
    Median100=-12.25 Variance100=3.6580250000000007
(Expected Sarsa) Episode 200: Length=43  Total return = -4.84 Energy Left = 0
    Median100=-12.25 Variance100=2.7068009899999996
(Expected Sarsa) Episode 300: Length=50  Total return = -12.25 Energy Left = 0
    Median100=-12.25 Variance100=5.47421716
(Expected Sarsa) Episode 400: Length=30  Total return = -8.75 Energy Left = 24
    Median100=-12.25 Variance100=39.94253499999999
(Expected Sarsa) Episode 500: Length=48  Total return = 7.96 Energy Left = 24
    Median100=-10.35 Variance100=60.34896475000001
(Expected Sarsa) Episode 600: Length=50  Total return = -12.25 Energy Left = 24
    Median100=-12.25 Variance100=22.318735039999996
(Expected Sarsa) Episode 700: Length=3  Total return = -10.24 Energy Left = 24
    Median100=-12.25 Variance100=2.0788000000000006
(Expected Sarsa) Episode 800: Length=50  Total return = -12.25 Energy Left = 24
    Median100=-12.25 Variance100=3.993488189999999
(Expected Sarsa) Episode 900: Length=50  Total return = -12.25 Energy Left = 24
    Median100=-12.25 Variance100=3.7693887499999983
(Expected Sarsa) Episode 1000: Length=50  Total return = -12.25 Energy Left = 24
    Median100=-12.25 Variance100=0.6536427499999995
(Expected Sarsa) Episode 1100: Length=50  Total return = -12.25 Energy Left = 24
    Median100=-12.25 Variance100=3.01200291
(Expected Sarsa) Episode 1200: Length=30  Total return = -8.55 Energy Left = 24
    Median100=-12.25 Variance100=4.256514760000001
(Expected Sarsa) Episode 1300: Length=30  Total return = 2.45 Energy Left = 28
    Median100=1.46 Variance100=13.662960509999998
(Expected Sarsa) Episode 1400: Length=21  Total return = 1.21 Energy Left = 28
    Median100=1.46 Variance100=14.89053859
(Expected Sarsa) Episode 1500: Length=33  Total return = 3.31 Energy Left = 28
```

```
    Median100=1.46 Variance100=13.472814110000002
(Expected Sarsa) Episode 1600: Length=38  Total return = 4.41 Energy Left = 28
    Median100=1.4 Variance100=16.637996750000003
(Expected Sarsa) Episode 1700: Length=27  Total return = 2.08 Energy Left = 29
    Median100=1.46 Variance100=14.226629839999998
(Expected Sarsa) Episode 1800: Length=25  Total return = 1.75 Energy Left = 30
    Median100=1.46 Variance100=15.04080019
(Expected Sarsa) Episode 1900: Length=23  Total return = 1.46 Energy Left = 30
    Median100=1.46 Variance100=12.26899211
(Expected Sarsa) Episode 2000: Length=21  Total return = 1.21 Energy Left = 30
    Median100=1.46 Variance100=11.714811710000001
(Expected Sarsa) Episode 2100: Length=25  Total return = 1.75 Energy Left = 30
    Median100=1.46 Variance100=12.136040190000001
(Expected Sarsa) Episode 2200: Length=23  Total return = 1.46 Energy Left = 30
    Median100=1.46 Variance100=9.74662396
(Expected Sarsa) Episode 2300: Length=22  Total return = 1.13 Energy Left = 30
    Median100=1.46 Variance100=16.50671475
(Expected Sarsa) Episode 2400: Length=21  Total return = 1.21 Energy Left = 30
    Median100=1.46 Variance100=14.58558211
(Expected Sarsa) Episode 2500: Length=25  Total return = 1.75 Energy Left = 30
    Median100=1.46 Variance100=19.090892999999998
(Expected Sarsa) Episode 2600: Length=27  Total return = 2.08 Energy Left = 30
    Median100=1.46 Variance100=13.494764240000002
(Expected Sarsa) Episode 2700: Length=28  Total return = 2.26 Energy Left = 30
    Median100=1.46 Variance100=10.853620109999998
(Expected Sarsa) Episode 2800: Length=24  Total return = 1.4 Energy Left = 30
    Median100=1.46 Variance100=13.41423136
(Expected Sarsa) Episode 2900: Length=23  Total return = 1.46 Energy Left = 30
    Median100=1.5550000000000002 Variance100=12.26003624
(Expected Sarsa) Episode 3000: Length=24  Total return = 1.6 Energy Left = 30
    Median100=1.46 Variance100=13.670378989999996
(Expected Sarsa) Episode 3100: Length=27  Total return = 2.08 Energy Left = 30
    Median100=1.46 Variance100=8.56510864
(Expected Sarsa) Episode 3200: Length=23  Total return = 1.46 Energy Left = 30
    Median100=1.46 Variance100=11.486330189999999
(Expected Sarsa) Episode 3300: Length=21  Total return = 1.21 Energy Left = 30
    Median100=1.46 Variance100=9.160533959999999
(Expected Sarsa) Episode 3400: Length=27  Total return = 1.6800000000000002
Energy Left = 30
    Median100=1.46 Variance100=10.12136675
(Expected Sarsa) Episode 3500: Length=22  Total return = -9.67 Energy Left = 30
    Median100=1.46 Variance100=5.85629371
(Expected Sarsa) Episode 3600: Length=28  Total return = 2.06 Energy Left = 30
    Median100=1.46 Variance100=13.136918749999998
(Expected Sarsa) Episode 3700: Length=23  Total return = 1.46 Energy Left = 30
    Median100=1.6 Variance100=7.564978999999999
(Expected Sarsa) Episode 3800: Length=27  Total return = 2.08 Energy Left = 30
    Median100=1.46 Variance100=11.39906611
```

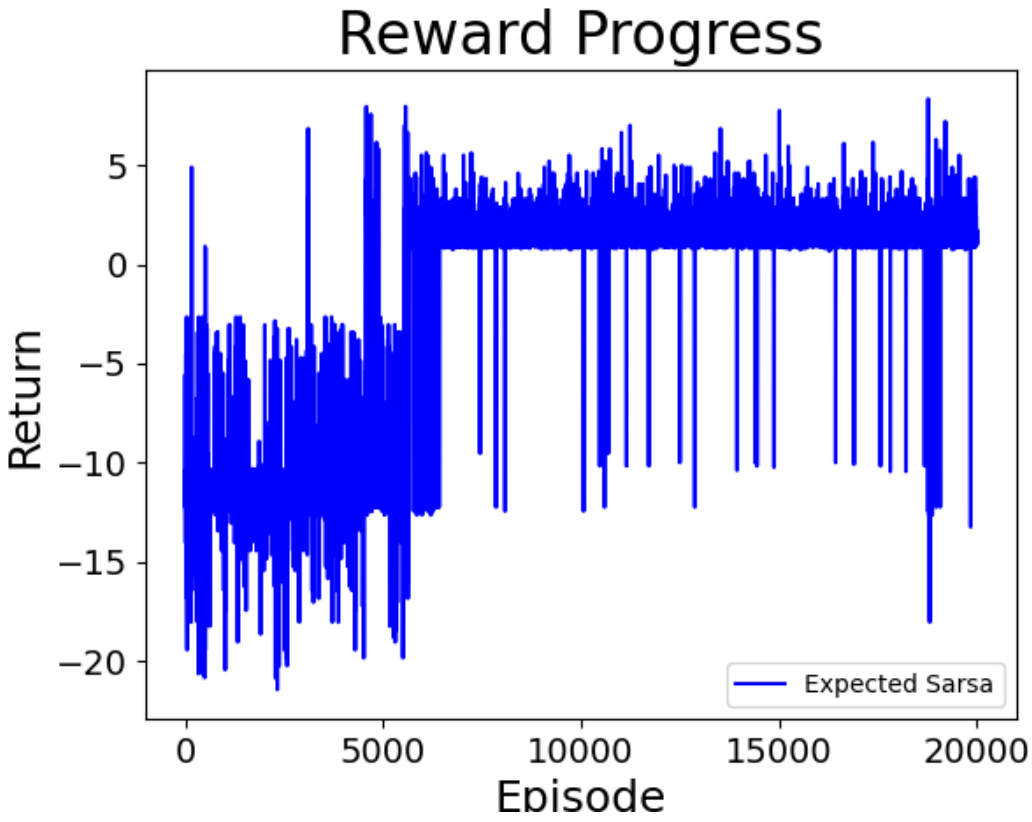
```
(Expected Sarsa) Episode 3900: Length=29  Total return = 2.45 Energy Left = 30
    Median100=1.46 Variance100=16.353733390000002
(Expected Sarsa) Episode 4000: Length=27  Total return = 2.08 Energy Left = 30
    Median100=1.46 Variance100=11.782940589999999
(Expected Sarsa) Episode 4100: Length=21  Total return = 1.21 Energy Left = 30
    Median100=1.46 Variance100=16.222031960000002
(Expected Sarsa) Episode 4200: Length=21  Total return = 1.21 Energy Left = 30
    Median100=1.46 Variance100=16.46123779
(Expected Sarsa) Episode 4300: Length=28  Total return = 2.06 Energy Left = 30
    Median100=1.46 Variance100=12.987120910000003
(Expected Sarsa) Episode 4400: Length=5  Total return = -10.35 Energy Left = 30
    Median100=1.46 Variance100=14.521776440000002
(Expected Sarsa) Episode 4500: Length=28  Total return = 2.06 Energy Left = 30
    Median100=1.46 Variance100=11.018604439999999
(Expected Sarsa) Episode 4600: Length=27  Total return = 2.08 Energy Left = 30
    Median100=1.46 Variance100=22.66003963999999
(Expected Sarsa) Episode 4700: Length=23  Total return = 1.46 Energy Left = 30
    Median100=1.46 Variance100=14.260672559999996
(Expected Sarsa) Episode 4800: Length=26  Total return = 1.71 Energy Left = 30
    Median100=1.46 Variance100=14.74025675
(Expected Sarsa) Episode 4900: Length=23  Total return = 1.46 Energy Left = 30
    Median100=1.655 Variance100=8.529964749999998
(Expected Sarsa) Episode 5000: Length=28  Total return = 2.26 Energy Left = 30
    Median100=1.46 Variance100=9.899934989999998
(Expected Sarsa) Episode 5100: Length=3  Total return = -10.24 Energy Left = 30
    Median100=1.46 Variance100=12.384772750000002
(Expected Sarsa) Episode 5200: Length=21  Total return = 1.21 Energy Left = 30
    Median100=1.46 Variance100=9.78351116
(Expected Sarsa) Episode 5300: Length=21  Total return = 1.21 Energy Left = 30
    Median100=1.46 Variance100=10.27099299
(Expected Sarsa) Episode 5400: Length=36  Total return = 3.8600000000000003
Energy Left = 30
    Median100=1.53 Variance100=7.2350778400000015
(Expected Sarsa) Episode 5500: Length=23  Total return = 1.46 Energy Left = 30
    Median100=1.46 Variance100=7.1540507600000005
(Expected Sarsa) Episode 5600: Length=25  Total return = 1.75 Energy Left = 30
    Median100=1.46 Variance100=7.650441360000002
(Expected Sarsa) Episode 5700: Length=24  Total return = 1.4 Energy Left = 30
    Median100=1.46 Variance100=14.400439159999998
(Expected Sarsa) Episode 5800: Length=26  Total return = 1.9100000000000001
Energy Left = 30
    Median100=1.46 Variance100=5.293603709999998
(Expected Sarsa) Episode 5900: Length=23  Total return = 1.46 Energy Left = 30
    Median100=1.53 Variance100=9.61707275
(Expected Sarsa) Episode 6000: Length=35  Total return = 3.8000000000000003
Energy Left = 30
    Median100=1.46 Variance100=13.39474416
(Expected Sarsa) Episode 6100: Length=23  Total return = 1.46 Energy Left = 30
```

```
    Median100=1.46 Variance100=6.0963426400000005
(Expected Sarsa) Episode 6200: Length=24  Total return = 1.6 Energy Left = 30
    Median100=1.46 Variance100=9.54470484
(Expected Sarsa) Episode 6300: Length=3  Total return = -10.24 Energy Left = 30
    Median100=1.46 Variance100=6.79041579
(Expected Sarsa) Episode 6400: Length=23  Total return = 1.46 Energy Left = 30
    Median100=1.46 Variance100=17.880320439999995
(Expected Sarsa) Episode 6500: Length=27  Total return = 2.08 Energy Left = 30
    Median100=1.46 Variance100=16.826134509999996
(Expected Sarsa) Episode 6600: Length=23  Total return = 1.46 Energy Left = 30
    Median100=1.46 Variance100=14.54420564
(Expected Sarsa) Episode 6700: Length=27  Total return = 2.08 Energy Left = 30
    Median100=1.46 Variance100=18.431422749999996
(Expected Sarsa) Episode 6800: Length=19  Total return = -10.0 Energy Left = 30
    Median100=1.46 Variance100=12.865790750000004
(Expected Sarsa) Episode 6900: Length=23  Total return = 1.46 Energy Left = 30
    Median100=1.46 Variance100=14.947930749999996
(Expected Sarsa) Episode 7000: Length=21  Total return = 1.21 Energy Left = 30
    Median100=1.46 Variance100=14.767018560000002
(Expected Sarsa) Episode 7100: Length=23  Total return = 1.46 Energy Left = 30
    Median100=1.46 Variance100=8.187099
(Expected Sarsa) Episode 7200: Length=23  Total return = 1.46 Energy Left = 30
    Median100=1.71 Variance100=8.379544590000002
(Expected Sarsa) Episode 7300: Length=21  Total return = 1.21 Energy Left = 30
    Median100=1.46 Variance100=8.594172839999999
(Expected Sarsa) Episode 7400: Length=25  Total return = 1.75 Energy Left = 30
    Median100=1.46 Variance100=18.452478840000005
(Expected Sarsa) Episode 7500: Length=29  Total return = 2.45 Energy Left = 30
    Median100=1.53 Variance100=5.366484840000001
(Expected Sarsa) Episode 7600: Length=23  Total return = 1.46 Energy Left = 30
    Median100=1.46 Variance100=11.624946110000002
(Expected Sarsa) Episode 7700: Length=23  Total return = 1.46 Energy Left = 30
    Median100=1.46 Variance100=16.723151960000006
(Expected Sarsa) Episode 7800: Length=22  Total return = 1.13 Energy Left = 30
    Median100=1.46 Variance100=13.144415839999999
(Expected Sarsa) Episode 7900: Length=27  Total return = 2.08 Energy Left = 30
    Median100=1.46 Variance100=8.37415436
(Expected Sarsa) Episode 8000: Length=22  Total return = 1.33 Energy Left = 30
    Median100=1.6 Variance100=3.40517475
(Expected Sarsa) Episode 8100: Length=27  Total return = 2.08 Energy Left = 30
    Median100=1.46 Variance100=10.400210439999999
(Expected Sarsa) Episode 8200: Length=4  Total return = -10.3 Energy Left = 30
    Median100=1.46 Variance100=13.36788491
(Expected Sarsa) Episode 8300: Length=25  Total return = 1.75 Energy Left = 30
    Median100=1.46 Variance100=14.866862999999999
(Expected Sarsa) Episode 8400: Length=21  Total return = 1.21 Energy Left = 30
    Median100=1.6400000000000001 Variance100=11.420383389999998
(Expected Sarsa) Episode 8500: Length=31  Total return = 2.86 Energy Left = 30
```

```
    Median100=1.46 Variance100=8.892673310000001
(Expected Sarsa) Episode 8600: Length=29  Total return = 2.45 Energy Left = 30
    Median100=1.46 Variance100=10.215174589999998
(Expected Sarsa) Episode 8700: Length=27  Total return = 2.08 Energy Left = 30
    Median100=1.4 Variance100=18.56801419
(Expected Sarsa) Episode 8800: Length=23  Total return = 1.46 Energy Left = 30
    Median100=1.46 Variance100=12.35408171
(Expected Sarsa) Episode 8900: Length=23  Total return = 1.46 Energy Left = 30
    Median100=1.46 Variance100=12.065832189999998
(Expected Sarsa) Episode 9000: Length=27  Total return = 2.08 Energy Left = 30
    Median100=1.46 Variance100=16.616547000000004
(Expected Sarsa) Episode 9100: Length=21  Total return = 1.21 Energy Left = 30
    Median100=1.46 Variance100=12.41883116
(Expected Sarsa) Episode 9200: Length=24  Total return = 1.4 Energy Left = 30
    Median100=1.46 Variance100=12.326871440000001
(Expected Sarsa) Episode 9300: Length=25  Total return = 1.75 Energy Left = 30
    Median100=1.46 Variance100=7.29771531
(Expected Sarsa) Episode 9400: Length=37  Total return = 4.33 Energy Left = 30
    Median100=1.46 Variance100=11.991128189999994
(Expected Sarsa) Episode 9500: Length=21  Total return = 1.21 Energy Left = 30
    Median100=1.46 Variance100=9.08105659
(Expected Sarsa) Episode 9600: Length=29  Total return = 2.45 Energy Left = 30
    Median100=1.46 Variance100=12.15457544
(Expected Sarsa) Episode 9700: Length=21  Total return = 1.21 Energy Left = 30
    Median100=1.43 Variance100=18.694577440000003
(Expected Sarsa) Episode 9800: Length=29  Total return = 2.45 Energy Left = 30
    Median100=1.46 Variance100=13.116796440000003
(Expected Sarsa) Episode 9900: Length=5  Total return = -10.35 Energy Left = 30
    Median100=1.46 Variance100=15.36803171
game over -- Algorithm Expected Sarsa completed

            [Expected Sarsa]: Max Reward = 8.35, MedLast100 = 1.46, VarLast100 =
10.101902750000002
```



The convergence of the algorithm is observed around 1000 episodes for first time. On further increasing the number of episodes to 20000, the agent gives same number of remaining energy without much deviation.



Saving the model

```
RL.save('./models/model.pkl')
```

```
# Reload and run the saved environment
env = Maze(agentXY, goalXY, walls=wall_shape, pits=pits, energy_capacity=50,␣
 ↪energy_factor=0.01)
RL = rlalgorithm(actions=list(range(env.n_actions)))
RL.load('./models/model.pkl')
env.run_without_learning(RL, episodes=10, sim_speed=0.001)
```

```
Using Expected Sarsa …
```