In [1]:

```
#importing necessary libraries
import numpy as np
import matplotlib.pyplot as plt
import random
from tqdm import tqdm
```

In [2]:

```
#building the environment
class Environment:
    def __init__(self, M,N, holes, terminal_state): #initializing the environment state
s, holes, terminals and rewards
        self.states = set()
        self.shape = (M,N)
        self.holes = holes
        self.terminal_state = terminal_state

        for i in range(1,M+1):
            for j in range(1, N+1):
                if (i,j) not in holes:
                    self.states.add((i,j))

        self.rewards = self.initialize_rewards()
        self.prob_agent_action = [0.8, 0.1, 0.05, 0.05]
    def initialize_rewards(self): #function to initialize the rewards for each state of
the environment
        r = {}
        for state in self.states:
            if state == (6,3):
                r[state] = -15
            elif state == (6,6):
                r[state] = 15
            else:
                r[state] = 0

        return r

    def agent_move(self, s, a): #function to update the state of the agent given an act
ion a and current state s
        x, y = s
        if a=='U':
            x = x-1
        elif a=='D':
            x = x + 1
        elif a=='R':
            y = y + 1
        elif a=='L':
            y = y - 1

        stay_same = self.check_corner_and_hole((x,y))

        if stay_same:
            return s

        return (x,y)

    def move_clockwise90(self, a): #function to return the action which is a 90 degree
 rotation to current action a
        if a=='U':
            return 'R'
        elif a=='R':
            return 'D'
        elif a=='D':
            return 'L'
        elif a=='L':
            return 'U'
```

```
    def move_anti_clockwise90(self, a): #function to return the action which is a 90 de
gree rotation to current action a
        if a=='U':
            return 'L'
        elif a=='L':
            return 'D'
        elif a=='D':
            return 'R'
        elif a=='R':
            return 'U'

    def check_corner_and_hole(self, s):
        #function to check if the updates state goes out of the gridworld or goes into
 holes.
        #If so, it returns a True value to address that the update should not take plac
e and agent should remain in current state.
        x1, y1 = s
        stay_same = False
        for hole in self.holes:
            if (x1,y1) == hole:
                stay_same = True

        if x1<1 or x1>6:
            stay_same = True
        if y1<1 or y1>6:
            stay_same = True

        return stay_same
```

In [3]:

```
gridworld = Environment(6,6, [(4,3),(5,3)], (6,6))
```

In [8]:

```
#building the agent
class Agent:
    def __init__(self, gamma, env, terminal_state_value = 0):
        #initializing the agent parameters
        self.actions = ['L','R','U','D'] #possible actions
        self.gamma = gamma #discount parameter
        self.terminal_state_value = terminal_state_value
        self.pi = self.initialize_policy(env)
        self.V = self.initialize_value_states(env)
        self.Q = self.initialize_Qvalue(env)


    def initialize_policy(self,env):
        #initializing the agent policy
        pi = {}
        for s in env.states:
            pi[s] = {}
            for a in self.actions:
                pi[s][a] = 0.25

        return pi

    def initialize_Qvalue(self, env):
        #initializing q values for the agent
        Q = {}
        for s in env.states:
            Q[s] = {}
            for a in self.actions:
                Q[s][a] = 0

        return Q

    def initialize_value_states(self, env):
        #initializing value of states
        v_s = {}
        for state in env.states:
            if state == env.terminal_state:
                #0 for part 1.a, 15 for for part 1.b)
                v_s[state] = self.terminal_state_value
            else:
                v_s[state] = random.random()
        return v_s

    def compute_value(self, env, epsilon = 1e-8):
        #function for policy evaluation to compute value of states and state-action pai
r Q values
        delta = np.inf

        while delta > epsilon:
            max_diff = -np.inf
            for s in env.states:
                if s!=env.terminal_state:
                    v = 0
                    for a in self.pi[s]:
                        p = self.pi[s][a]
                        #with prob 0.8 take action a to get to state s1 with reward r1
                        s1 = env.agent_move(s,a)
                        r1 = env.rewards[s1]
```

```
                        v1 = 0.8 * (r1 + self.gamma*self.V[s1])

                        #with prob 0.1 stay in same state s
                        r2 = env.rewards[s]
                        v2 = 0.1 * (r2 + self.gamma * self.V[s])

                        #with prob 0.05 take action in direction +90 degree clockwise d
irection

                        a3 = env.move_clockwise90(a)
                        s3 = env.agent_move(s, a3)
                        r3 = env.rewards[s3]
                        v3 = 0.05 * (r3 + self.gamma * self.V[s3])

                        #with prob 0.05 take action in direction -90 degree clockwise d
irection


                        a4 = env.move_anti_clockwise90(a)
                        s4 = env.agent_move(s, a4)
                        r4 = env.rewards[s4]
                        v4 = 0.05 * (r4 + self.gamma * self.V[s4])

                        v = v + (p * (v1+v2+v3+v4))
                        self.Q[s][a] = (v1 + v2 + v3 + v4)

                    diff = abs(self.V[s] - v)
                    self.V[s] = v
                    max_diff = max(max_diff, diff)

            delta = max_diff

    def policy_improve(self, env):
        #function for policy improvement where the policy is improved greedily

        for s in env.states:
            if s!=env.terminal_state:
                greedy_actions = []
                max_q = -np.inf

                for a in self.pi[s]:
                    #if self.pi[s][a]>0:
                    s1 = env.agent_move(s,a)
                    r1 = env.rewards[s1]

                    v1 = 0.8 * (r1 + self.gamma*self.V[s1])

                    #with prob 0.1 stay in same state s
                    r2 = env.rewards[s]
                    v2 = 0.1 * (r2 + self.gamma * self.V[s])

                    #with prob 0.05 take action in direction +90 degree clockwise direc
tion
                    a3 = env.move_clockwise90(a)
                    s3 = env.agent_move(s, a3)
                    r3 = env.rewards[s3]
                    v3 = 0.05 * (r3 + self.gamma * self.V[s3])

                    #with prob 0.05 take action in direction -90 degree clockwise direc
tion


                    a4 = env.move_anti_clockwise90(a)
                    s4 = env.agent_move(s, a4)
```

```
                r4 = env.rewards[s4]
                v4 = 0.05 * (r4 + self.gamma * self.V[s4])

                v = (v1 + v2 + v3 + v4)

                if v > max_q:
                    greedy_actions = [a]
                    max_q = v
                elif v == max_q:
                    greedy_actions.append(a)

            #update policy greedily
            n = len(greedy_actions)
            for a in self.pi[s]:
                if a in greedy_actions:
                    self.pi[s][a] = 1.0/n
                else:
                    self.pi[s][a] = 0


    def policy_iterate(self, env, iterations = 100):
        #function for policy iteration
        v_11_across_ep = []
        for iteration_number in range(iterations):
            self.compute_value(env)
            v_11_across_ep.append(self.V[(1,1)])
            self.policy_improve(env)

        return v_11_across_ep

    def find_optimal_move(self, s):
        #function to find the optimal move at a given state s
        optimal_moves = []
        max_v = 0
        for a in self.pi[s]:
            if self.pi[s][a] > max_v:
                optimal_moves = [a]
                max_v = self.pi[s][a]
            elif self.pi[s][a] == max_v:
                optimal_moves.append(a)

        return optimal_moves


    def plot_state_and_policy(self, env, plot_state_values = False, label = None):
        #function to plot state values and policy
        plt.figure(figsize = (20, 10))
        Grid_plot=plt.subplot()
        M,N = env.shape
        for i in range(M):
            for j in range(N):
                s = (i+1, j+1)
                if s==env.terminal_state:
                    if plot_state_values:
                        t = round(self.V[s], 3)
                        value = str(s) + "\n\n" + "TERMINAL"
                    else:
                        value = "TERMINAL"
                elif s not in env.holes:
                    value=str(s)
                    move = self.find_optimal_move(s)
```

```
                if plot_state_values==False:
                    value = value + '\n\n' + ','.join(move)
                else:
                    t = round(self.V[s],3)
                    value = value + '\n\n' + str(t)
            else:
                value = "HOLE"



            Grid_plot.text(j+0.5,N-i-0.5,value,ha='center',va='center')

        Grid_plot.grid(color='k')
        Grid_plot.axis('scaled')
        Grid_plot.axis([0, M, 0, N])

        Grid_plot.set_yticklabels([])
        Grid_plot.set_xticklabels([])
        if label is not None:
            plt.savefig(label)
```

1.a)

i) Value of states for random policy of taking each action with probability 0.25

In [9]:

```
agent = Agent(0.9, gridworld)
agent.compute_value(gridworld)
agent.plot_state_and_policy(gridworld, plot_state_values = True,label = 'plots/1_a.png'
)
```

(1, 1) -0.471	(1, 2) -0.406	(1, 3) -0.276	(1, 4) -0.15	(1, 5) -0.022	(1, 6) 0.067
(2, 1) -0.768	(2, 2) -0.673	(2, 3) -0.409	(2, 4) -0.227	(2, 5) 0.006	(2, 6) 0.19
(3, 1) -1.54	(3, 2) -1.44	(3, 3) -0.662	(3, 4) -0.465	(3, 5) 0.084	(3, 6) 0.592
(4, 1) -3.174	(4, 2) -3.594	HOLE	(4, 4) -1.284	(4, 5) 0.245	(4, 6) 1.793
(5, 1) -5.954	(5, 2) -7.943	HOLE	(5, 4) -4.266	(5, 5) 0.507	(5, 6) 5.429
(6, 1) -9.687	(6, 2) -18.203	(6, 3) -29.301	(6, 4) -14.128	(6, 5) 0.872	(6, 6) TERMINAL

In [10]:

```
def plot_q_values(agent, env, label = None):
    plt.figure(figsize = (20, 10))
    Grid_plot=plt.subplot()
    M,N = env.shape
    for i in range(M):
        for j in range(N):
            s = (i+1, j+1)
            if s==env.terminal_state:
                value = str(s) + "\n\n" + "TERMINAL"
            elif s not in env.holes:
                value=str(s)
                qs = {}
                for a in agent.actions:
                    qs[a] = round(agent.Q[s][a],3)

                value = value + "\n\n" + "L :  {}\nR :  {}\nU :  {}\nD :  {}".format(qs
['L'],qs['R'],qs['U'],qs['D'])
            else:
                value = "HOLE"



            Grid_plot.text(j+0.5,N-i-0.5,value,ha='center',va='center')

    Grid_plot.grid(color='k')
    Grid_plot.axis('scaled')
    Grid_plot.axis([0, M, 0, N])

    Grid_plot.set_yticklabels([])
    Grid_plot.set_xticklabels([])
    if label is not None:
        plt.savefig(label)
```

ii) Q-Values of each state action pair for the random policy of choosing each action with probability 0.25

In [11]:

```
plot_q_values(agent, gridworld, label = "plots/1_a_q_values.png")
```

(1, 1) L : -0.437 R : -0.391 U : -0.421 D : -0.635	(1, 2) L : -0.424 R : -0.284 U : -0.363 D : -0.555	(1, 3) L : -0.348 R : -0.164 U : -0.249 D : -0.344	(1, 4) L : -0.23 R : -0.046 U : -0.135 D : -0.19	(1, 5) L : -0.111 R : 0.046 U : -0.022 D : -0.002	(1, 6) L : 0.002 R : 0.066 U : 0.057 D : 0.145
(2, 1) L : -0.713 R : -0.644 U : -0.473 D : -1.243	(2, 2) L : -0.697 R : -0.438 U : -0.406 D : -1.15	(2, 3) L : -0.563 R : -0.242 U : -0.276 D : -0.554	(2, 4) L : -0.343 R : -0.044 U : -0.147 D : -0.373	(2, 5) L : -0.16 R : 0.14 U : -0.017 D : 0.059	(2, 6) L : 0.051 R : 0.184 U : 0.075 D : 0.452
(3, 1) L : -1.425 R : -1.352 U : -0.826 D : -2.558	(3, 2) L : -1.431 R : -0.798 U : -0.713 D : -2.816	(3, 3) L : -1.144 R : -0.442 U : -0.44 D : -0.622	(3, 4) L : -0.587 R : -0.049 U : -0.231 D : -0.992	(3, 5) L : -0.316 R : 0.445 U : 0.017 D : 0.19	(3, 6) L : 0.203 R : 0.569 U : 0.221 D : 1.375
(4, 1) L : -2.908 R : -3.21 U : -1.699 D : -4.877	(4, 2) L : -3.031 R : -3.333 U : -1.664 D : -6.347	HOLE	(4, 4) L : -1.253 R : -0.152 U : -0.497 D : -3.234	(4, 5) L : -0.876 R : 1.34 U : 0.105 D : 0.41	(4, 6) L : 0.609 R : 1.723 U : 0.679 D : 4.162
(5, 1) L : -5.402 R : -6.833 U : -3.446 D : -8.136	(5, 2) L : -5.983 R : -7.415 U : -3.928 D : -14.446	HOLE	(5, 4) L : -4.149 R : -0.712 U : -1.478 D : -10.726	(5, 5) L : -2.976 R : 4.005 U : 0.274 D : 0.726	(5, 6) L : 1.684 R : 5.228 U : 2.047 D : 12.756
(6, 1) L : -8.55 R : -14.682 U : -6.414 D : -9.101	(6, 2) L : -9.789 R : -35.912 U : -9.862 D : -17.249	(6, 3) L : -21.38 R : -18.447 U : -38.689 D : -38.689	(6, 4) L : -35.196 R : -1.472 U : -6.373 D : -13.473	(6, 5) L : -10.032 R : 12.14 U : 0.558 D : 0.82	(6, 6) TERMINAL

1.b) Policy Iteration

In [12]:

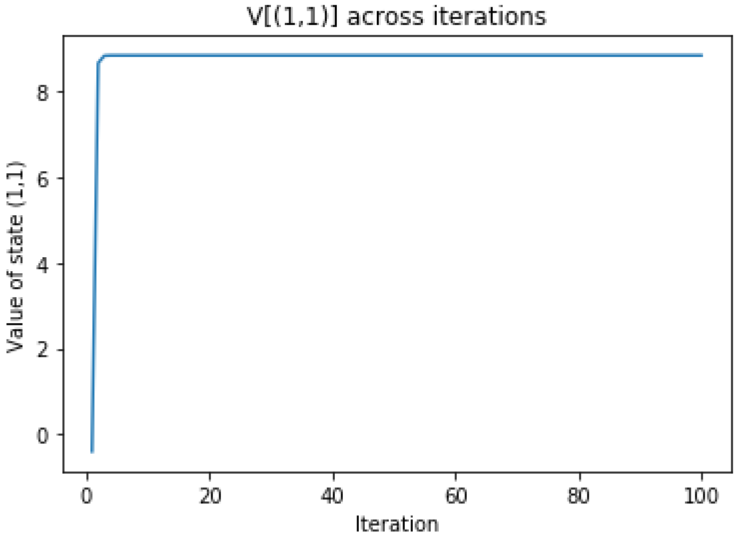
```
agent = Agent(0.9, gridworld, terminal_state_value = 15)
values = agent.policy_iterate(gridworld)
```

Plot of values of state (1,1) across iterations

In [13]:

```
def plot_values(values,label = None):
        ts = np.arange(1,(len(values)+1))
        plt.plot(ts, values)
        plt.xlabel("Iteration")
        plt.ylabel("Value of state (1,1)")
        plt.title("V[(1,1)] across iterations")
        if label is not None:
            plt.savefig(label)
        plt.show()

plot_values(values, label = 'plots/1_b_value_state_across_iterations.png')
```



Plot of optimal policy for each state

In [14]:

```
agent.plot_state_and_policy(gridworld, label = 'plots/1_b_opt_policy.png')
```

(1, 1) R	(1, 2) R	(1, 3) R	(1, 4) D	(1, 5) D	(1, 6) D
(2, 1) R	(2, 2) R	(2, 3) R	(2, 4) D	(2, 5) D	(2, 6) D
(3, 1) R	(3, 2) R	(3, 3) R	(3, 4) D	(3, 5) D	(3, 6) D
(4, 1) U	(4, 2) U	HOLE	(4, 4) D	(4, 5) D	(4, 6) D
(5, 1) U	(5, 2) U	HOLE	(5, 4) R	(5, 5) R,D	(5, 6) D
(6, 1) U	(6, 2) U	(6, 3) R	(6, 4) R	(6, 5) R	TERMINAL

In [ ]:

```

```