CE/CZ4046
INTELLGENT AGENTS

Assignment 1

Sin Hui, Pamela (K1920110A)

Assignment 1 is implemented in Python.
Note that asynchronous utility value updates is being implemented.

Part 1

There are four ipynb notebooks provided:

Name of ipynb notebook	**Description**	**Key facts**
confirm_textbook_values.ipynb	Pre-cursor check that the code can produce utilities in the textbook	• Terminal states of reward -1 and +1 • Reward for non-terminal states is -0.04 • Discount factor of 1
confirm_assignment_values_gamma0.946.ipynb	Pre-cursor check that the code can produce the reference utilities in the Assignment 1 question handout	• No terminal states • Reward at different squares are +1, -1, -0.04 • Discount factor of 0.946
assignment_answer_part1.ipynb	Answer to Assignment 1, Part 1	• No terminal states • Reward at different squares are +1, -1, -0.04 • Discount factor of 0.99
assignment_answer_part2.ipynb	Answer to Assignment 1, Part 2	• No terminal states • Reward at different squares are +1, -1, -0.04 • Reward at (0,0) is randomly 0 or 1 for each iteration • Discount factor of 0.99

Note that for the below answers:

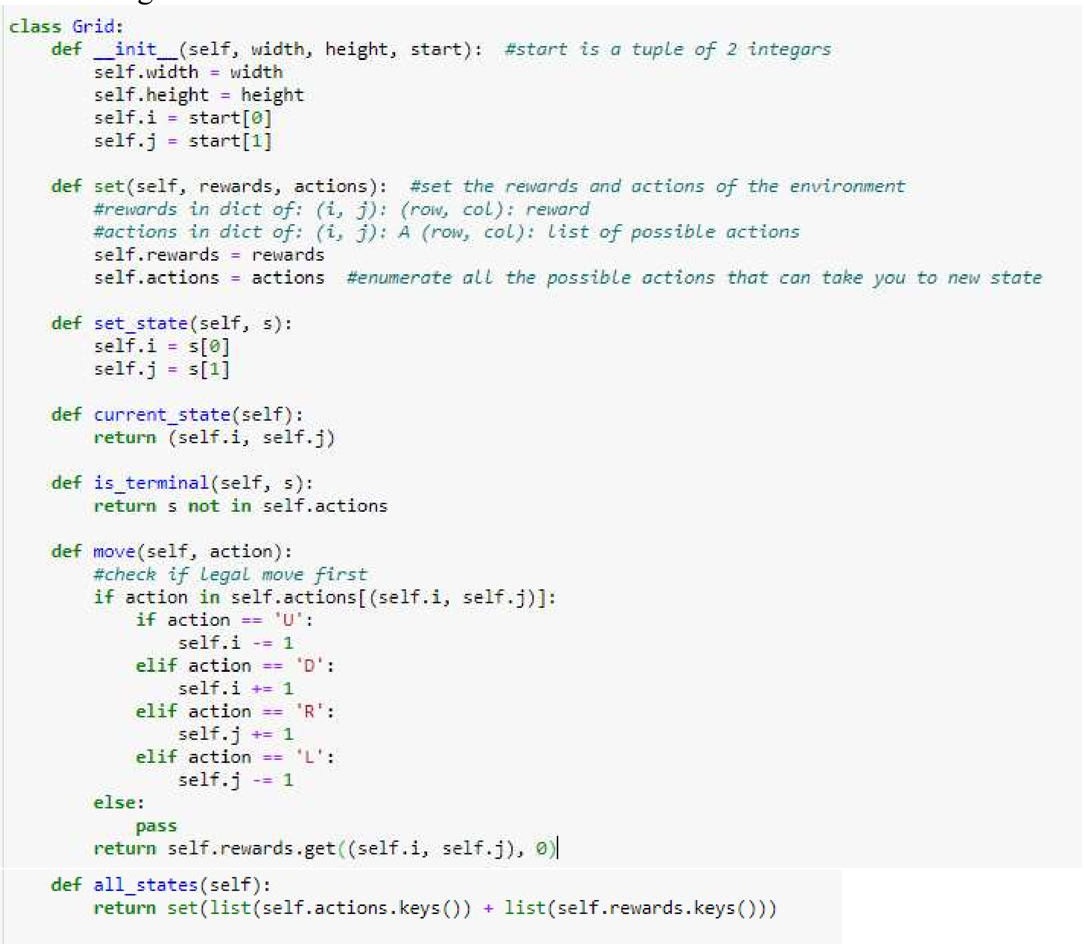
- Key:
  U: Up
  R: Right
  D: Down
  L: Left

- Note that for (ii), (iii) and (iv) below, coordinates are in (col,row) format with the top left corner being (0,0) as specified in the assignment.

Section 1: Value Iteration

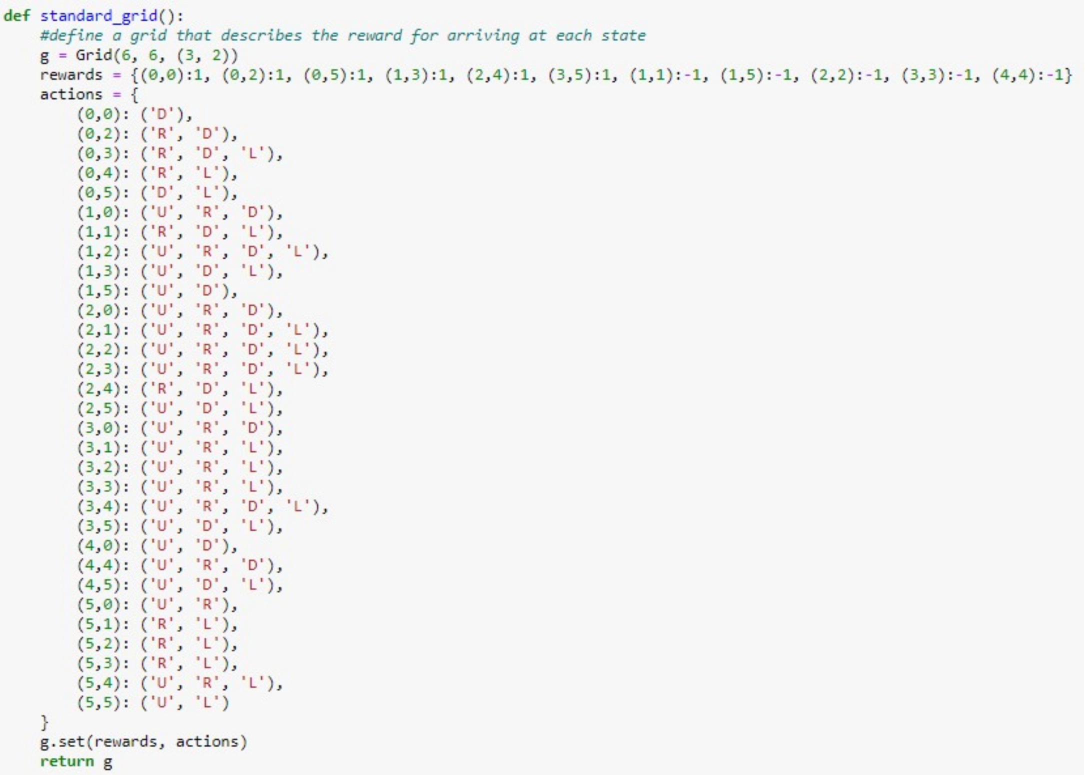
(i) Descriptions of implemented solutions

- State space *S*
  Define the state space of all possible configurations of the game such as the size of the grid world.

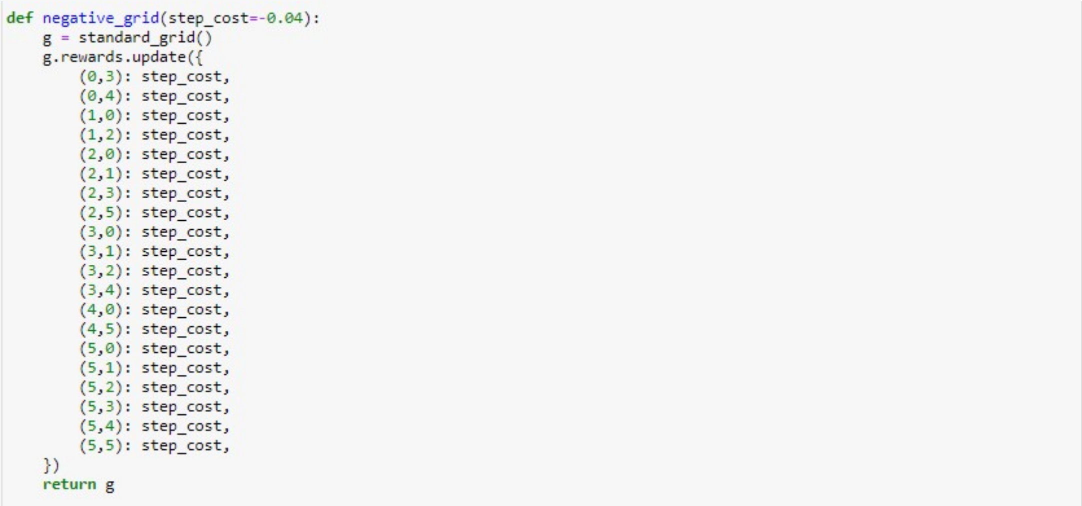


```
class Grid:
    def __init__(self, width, height, start):  #start is a tuple of 2 integars
        self.width = width
        self.height = height
        self.i = start[0]
        self.j = start[1]

    def set(self, rewards, actions):  #set the rewards and actions of the environment
        #rewards in dict of: (i, j): (row, col): reward
        #actions in dict of: (i, j): A (row, col): list of possible actions
        self.rewards = rewards
        self.actions = actions  #enumerate all the possible actions that can take you to new state

    def set_state(self, s):
        self.i = s[0]
        self.j = s[1]

    def current_state(self):
        return (self.i, self.j)

    def is_terminal(self, s):
        return s not in self.actions

    def move(self, action):
        #check if legal move first
        if action in self.actions[(self.i, self.j)]:
            if action == 'U':
                self.i -= 1
            elif action == 'D':
                self.i += 1
            elif action == 'R':
                self.j += 1
            elif action == 'L':
                self.j -= 1
        else:
            pass
        return self.rewards.get((self.i, self.j), 0)

    def all_states(self):
        return set(list(self.actions.keys()) + list(self.rewards.keys()))
```

- Action space A
  For each non-wall state, list possible legal actions (U, R, D, L) that bring the agent to the next state.



```
def standard_grid():
    #define a grid that describes the reward for arriving at each state
    g = Grid(6, 6, (3, 2))
    rewards = {(0,0):1, (0,2):1, (0,5):1, (1,3):1, (2,4):1, (3,5):1, (1,1):-1, (1,5):-1, (2,2):-1, (3,3):-1, (4,4):-1}
    actions = {
        (0,0): ('D'),
        (0,2): ('R', 'D'),
        (0,3): ('R', 'D', 'L'),
        (0,4): ('R', 'L'),
        (0,5): ('D', 'L'),
        (1,0): ('U', 'R', 'D'),
        (1,1): ('R', 'D', 'L'),
        (1,2): ('U', 'R', 'D', 'L'),
        (1,3): ('U', 'D', 'L'),
        (1,5): ('U', 'D'),
        (2,0): ('U', 'R', 'D'),
        (2,1): ('U', 'R', 'D', 'L'),
        (2,2): ('U', 'R', 'D', 'L'),
        (2,3): ('U', 'R', 'D', 'L'),
        (2,4): ('R', 'D', 'L'),
        (2,5): ('U', 'D', 'L'),
        (3,0): ('U', 'R', 'D'),
        (3,1): ('U', 'R', 'L'),
        (3,2): ('U', 'R', 'L'),
        (3,3): ('U', 'R', 'L'),
        (3,4): ('U', 'R', 'D', 'L'),
        (3,5): ('U', 'D', 'L'),
        (4,0): ('U', 'D'),
        (4,4): ('U', 'R', 'D'),
        (4,5): ('U', 'D', 'L'),
        (5,0): ('U', 'R'),
        (5,1): ('R', 'L'),
        (5,2): ('R', 'L'),
        (5,3): ('R', 'L'),
        (5,4): ('U', 'R', 'L'),
        (5,5): ('U', 'L')
    }
    g.set(rewards, actions)
    return g
```



```
def negative_grid(step_cost=-0.04):
    g = standard_grid()
    g.rewards.update({
        (0,3): step_cost,
        (0,4): step_cost,
        (1,0): step_cost,
        (1,2): step_cost,
        (2,0): step_cost,
        (2,1): step_cost,
        (2,3): step_cost,
        (2,5): step_cost,
        (3,0): step_cost,
        (3,1): step_cost,
        (3,2): step_cost,
        (3,4): step_cost,
        (4,0): step_cost,
        (4,5): step_cost,
        (5,0): step_cost,
        (5,1): step_cost,
        (5,2): step_cost,
        (5,3): step_cost,
        (5,4): step_cost,
        (5,5): step_cost,
    })
    return g
```

- Transition function $P(s' \mid s, \pi_i(s))$
  Define the probability of being in state s, taking action a, and ending up in state s'.
  Set if-else conditions such that the agent has a 0.8 probability of going in the intended direction, 0.1 probability of going in a direction 90 degrees clockwise of the intended direction, and 0.1 probability of going in a direction 90 degree anticlockwise of the intended direction.
  For example, if the agent's intended direction is R, there is a 0.8 probability of going R, 0.1 probability of going 90 degrees clockwise which is D, and a 0.1 probability of going 90 degrees anticlockwise which is U.

  Define that if the agent tries to do an action that is not in the list of possible defined legal actions (hits a wall or goes out of the boundary of the state space) for the state, the agent remains in the same state.

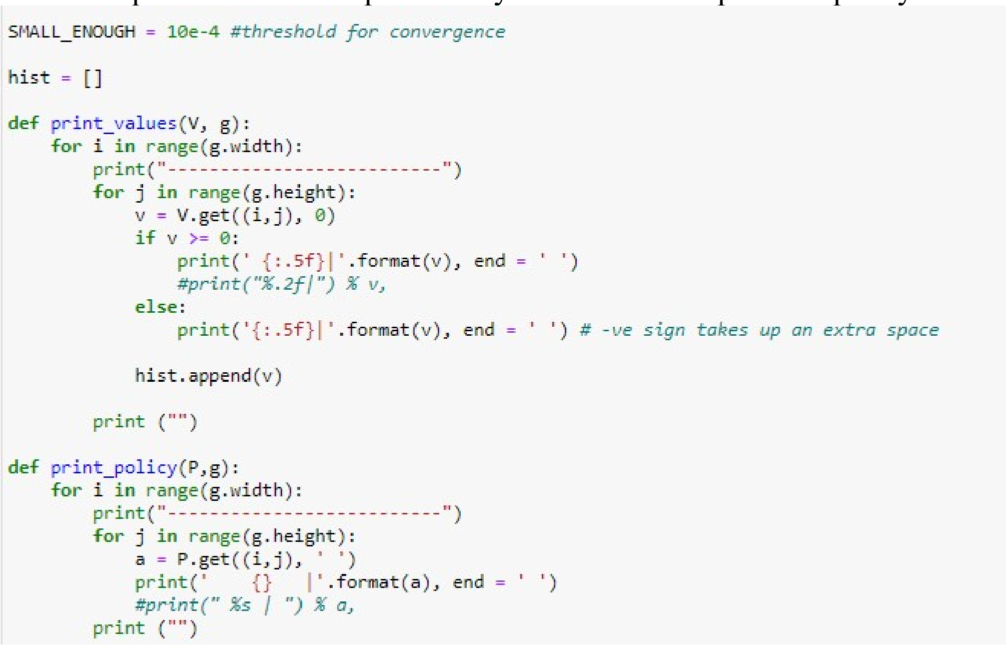
- Reward function
  Construct a grid that has the specified rewards at different states.

- Utilities of a state using the Bellman equation

  $$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) U(s')$$

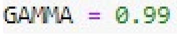
  Define that the utility of a state is the immediate reward for that state plus the expected discounted utility of the next state, assuming that the agent chooses the optimal action.

- Define helper functions that print utility values and that print the policy.



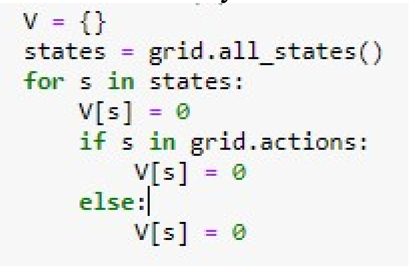
```
SMALL_ENOUGH = 10e-4 #threshold for convergence

hist = []

def print_values(V, g):
    for i in range(g.width):
        print("--------------------------")
        for j in range(g.height):
            v = V.get((i,j), 0)
            if v >= 0:
                print(' {:.5f}|'.format(v), end = ' ')
                #print("%.2f|") % v,
            else:
                print('{:.5f}|'.format(v), end = ' ') # -ve sign takes up an extra space

            hist.append(v)

        print ("")

def print_policy(P,g):
    for i in range(g.width):
        print("--------------------------")
        for j in range(g.height):
            a = P.get((i,j), ' ')
            print('    {}    |'.format(a), end = ' ')
            #print(" %s | ") % a,
        print ("")
```

- Set the discount factor gamma to 0.99.



```
GAMMA = 0.99
```

- Initialise all utility values to 0.

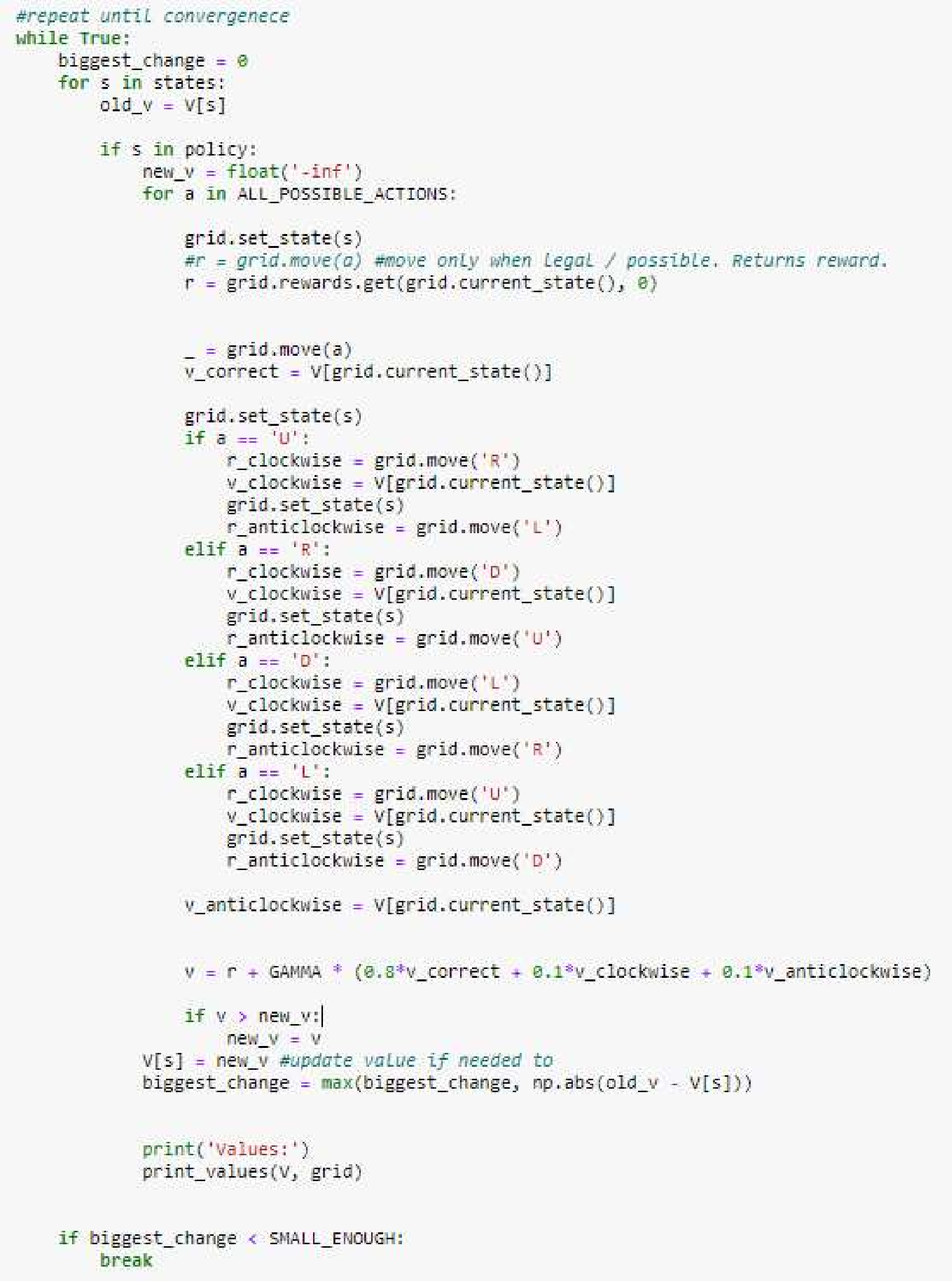


```
V = {}
states = grid.all_states()
for s in states:
    V[s] = 0
    if s in grid.actions:
        V[s] = 0
    else:
        V[s] = 0
```

- For i = 0, 1, 2… :

  Perform Bellman update (iteration steps)

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) U_i(s')$$

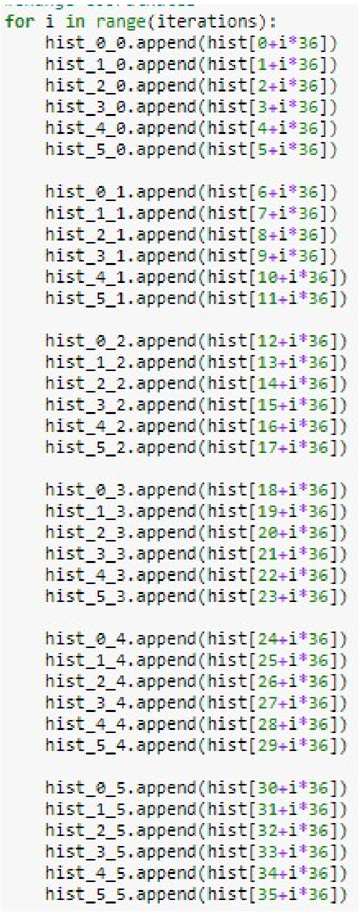


```
#repeat until convergenece
while True:
    biggest_change = 0
    for s in states:
        old_v = V[s]

        if s in policy:
            new_v = float('-inf')
            for a in ALL_POSSIBLE_ACTIONS:

                grid.set_state(s)
                #r = grid.move(a) #move only when legal / possible. Returns reward.
                r = grid.rewards.get(grid.current_state(), 0)


                _ = grid.move(a)
                v_correct = V[grid.current_state()]

                grid.set_state(s)
                if a == 'U':
                    r_clockwise = grid.move('R')
                    v_clockwise = V[grid.current_state()]
                    grid.set_state(s)
                    r_anticlockwise = grid.move('L')
                elif a == 'R':
                    r_clockwise = grid.move('D')
                    v_clockwise = V[grid.current_state()]
                    grid.set_state(s)
                    r_anticlockwise = grid.move('U')
                elif a == 'D':
                    r_clockwise = grid.move('L')
                    v_clockwise = V[grid.current_state()]
                    grid.set_state(s)
                    r_anticlockwise = grid.move('R')
                elif a == 'L':
                    r_clockwise = grid.move('U')
                    v_clockwise = V[grid.current_state()]
                    grid.set_state(s)
                    r_anticlockwise = grid.move('D')

                v_anticlockwise = V[grid.current_state()]


                v = r + GAMMA * (0.8*v_correct + 0.1*v_clockwise + 0.1*v_anticlockwise)

                if v > new_v:
                    new_v = v
            V[s] = new_v #update value if needed to
            biggest_change = max(biggest_change, np.abs(old_v - V[s]))


            print('Values:')
            print_values(V, grid)


    if biggest_change < SMALL_ENOUGH:
        break
```

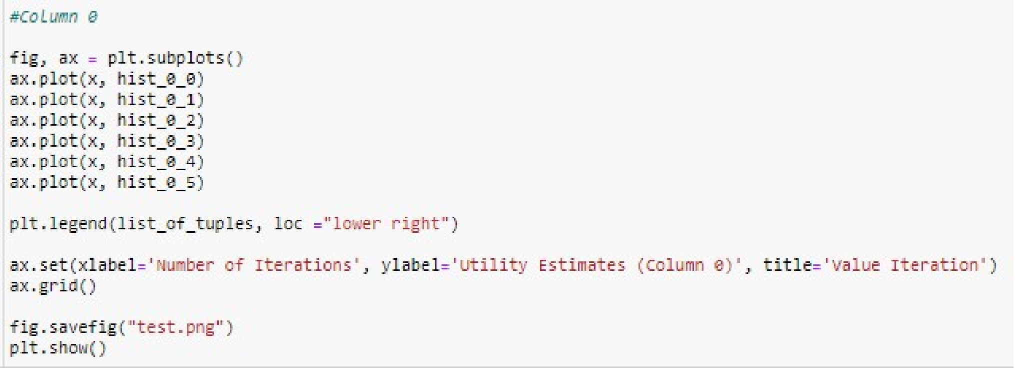
- Set the condition that for an iteration, if the biggest change in utility values for a state is less than 0.0001, consider that the utility values have converged.

- After utility values have converged, find a policy that leads to the converged utility values.

- Save the history of all values in a list.
  After the utility values have converged, split the saved history of values for each state separately.
  Plot the graphs of Utility Estimation against Number of Iterations for each state onto a single plot.
  Given that there are 36 multiple graphs for each of the 36 states, further plot 6 more plots, with each plot showing the graphs for states in the same column.

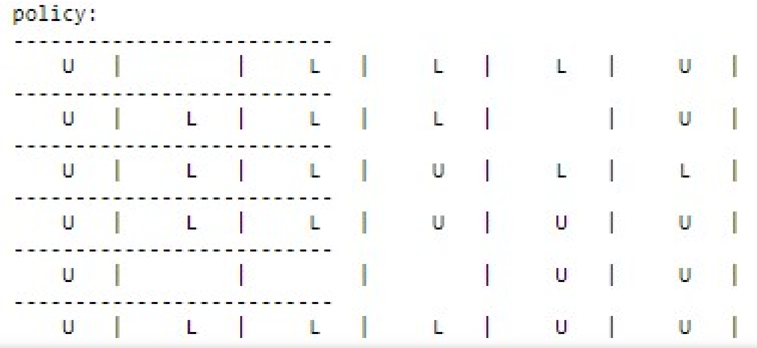


```
for i in range(iterations):
    hist_0_0.append(hist[0+i*36])
    hist_1_0.append(hist[1+i*36])
    hist_2_0.append(hist[2+i*36])
    hist_3_0.append(hist[3+i*36])
    hist_4_0.append(hist[4+i*36])
    hist_5_0.append(hist[5+i*36])

    hist_0_1.append(hist[6+i*36])
    hist_1_1.append(hist[7+i*36])
    hist_2_1.append(hist[8+i*36])
    hist_3_1.append(hist[9+i*36])
    hist_4_1.append(hist[10+i*36])
    hist_5_1.append(hist[11+i*36])

    hist_0_2.append(hist[12+i*36])
    hist_1_2.append(hist[13+i*36])
    hist_2_2.append(hist[14+i*36])
    hist_3_2.append(hist[15+i*36])
    hist_4_2.append(hist[16+i*36])
    hist_5_2.append(hist[17+i*36])

    hist_0_3.append(hist[18+i*36])
    hist_1_3.append(hist[19+i*36])
    hist_2_3.append(hist[20+i*36])
    hist_3_3.append(hist[21+i*36])
    hist_4_3.append(hist[22+i*36])
    hist_5_3.append(hist[23+i*36])

    hist_0_4.append(hist[24+i*36])
    hist_1_4.append(hist[25+i*36])
    hist_2_4.append(hist[26+i*36])
    hist_3_4.append(hist[27+i*36])
    hist_4_4.append(hist[28+i*36])
    hist_5_4.append(hist[29+i*36])

    hist_0_5.append(hist[30+i*36])
    hist_1_5.append(hist[31+i*36])
    hist_2_5.append(hist[32+i*36])
    hist_3_5.append(hist[33+i*36])
    hist_4_5.append(hist[34+i*36])
    hist_5_5.append(hist[35+i*36])
```



```
#Column 0

fig, ax = plt.subplots()
ax.plot(x, hist_0_0)
ax.plot(x, hist_0_1)
ax.plot(x, hist_0_2)
ax.plot(x, hist_0_3)
ax.plot(x, hist_0_4)
ax.plot(x, hist_0_5)

plt.legend(list_of_tuples, loc ="lower right")

ax.set(xlabel='Number of Iterations', ylabel='Utility Estimates (Column 0)', title='Value Iteration')
ax.grid()

fig.savefig("test.png")
plt.show()
```

(ii) Plot of optimal policy



```
policy:
--------------------------
   U  |     |   L  |   L  |   L  |   U  |
--------------------------
   U  |  L  |   L  |   L  |      |   U  |
--------------------------
   U  |  L  |   L  |   U  |   L  |   L  |
--------------------------
   U  |  L  |   L  |   U  |   U  |   U  |
--------------------------
   U  |     |      |      |   U  |   U  |
--------------------------
   U  |  L  |   L  |   L  |   U  |   U  |
```

Policy at (0,0): Up
Policy at (0,1): Up
Policy at (0,2): Up
Policy at (0,3): Up
Policy at (0,4): Up
Policy at (0,5): Up

Policy at (1,1): Left
Policy at (1,2): Left
Policy at (1,3): Left
Policy at (1,5): Left

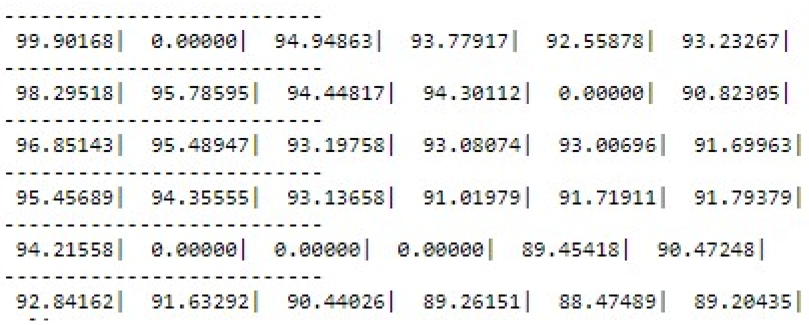
Policy at (2,0): Left
Policy at (2,1): Left
Policy at (2,2): Left
Policy at (2,3): Left
Policy at (2,5): Left

Policy at (3,0): Left
Policy at (3,1): Left
Policy at (3,2): Up
Policy at (3,3): Up
Policy at (3,5): Left

Policy at (4,0): Left
Policy at (4,2): Left
Policy at (4,3): Up
Policy at (4,4): Up
Policy at (4,5): Up

Policy at (5,0): Up
Policy at (5,1): Up
Policy at (5,2): Left
Policy at (5,3): Up
Policy at (5,4): Up
Policy at (5,5): Up

(iii) Utilities of all states



```
--------------------------
 99.90168|  0.00000|  94.94863|  93.77917|  92.55878|  93.23267|
--------------------------
 98.29518|  95.78595|  94.44817|  94.30112|  0.00000|  90.82305|
--------------------------
 96.85143|  95.48947|  93.19758|  93.08074|  93.00696|  91.69963|
--------------------------
 95.45689|  94.35555|  93.13658|  91.01979|  91.71911|  91.79379|
--------------------------
 94.21558|  0.00000|  0.00000|  0.00000|  89.45418|  90.47248|
--------------------------
 92.84162|  91.63292|  90.44026|  89.26151|  88.47489|  89.20435|
```

Reference utilities of states:

(0,0): 99.90168
(0,1): 98.29518
(0,2): 96.85143
(0,3): 95.45689
(0,4): 94.21558
(0,5): 92.84162

(1,1): 95.78595
(1,2): 95.48947
(1,3): 94.35555
(1,5): 91.63292

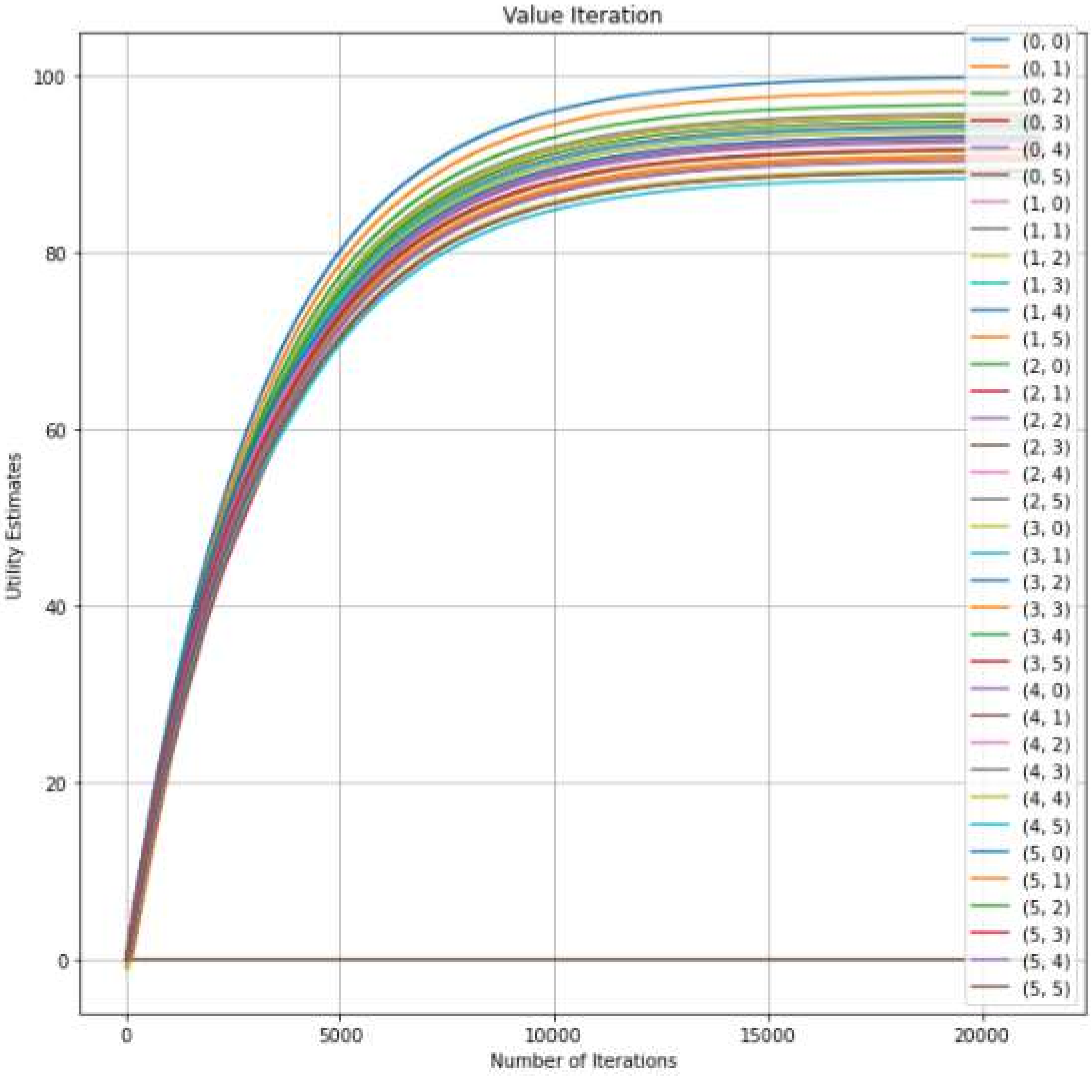
(2,0): 94.94863
(2,1): 94.44817
(2,2): 93.19758
(2,3): 93.13658
(2,5): 90.44026

(3,0): 93.77917
(3,1): 94.30112
(3,2): 93.08074
(3,3): 91.01979
(3,5): 89.26151

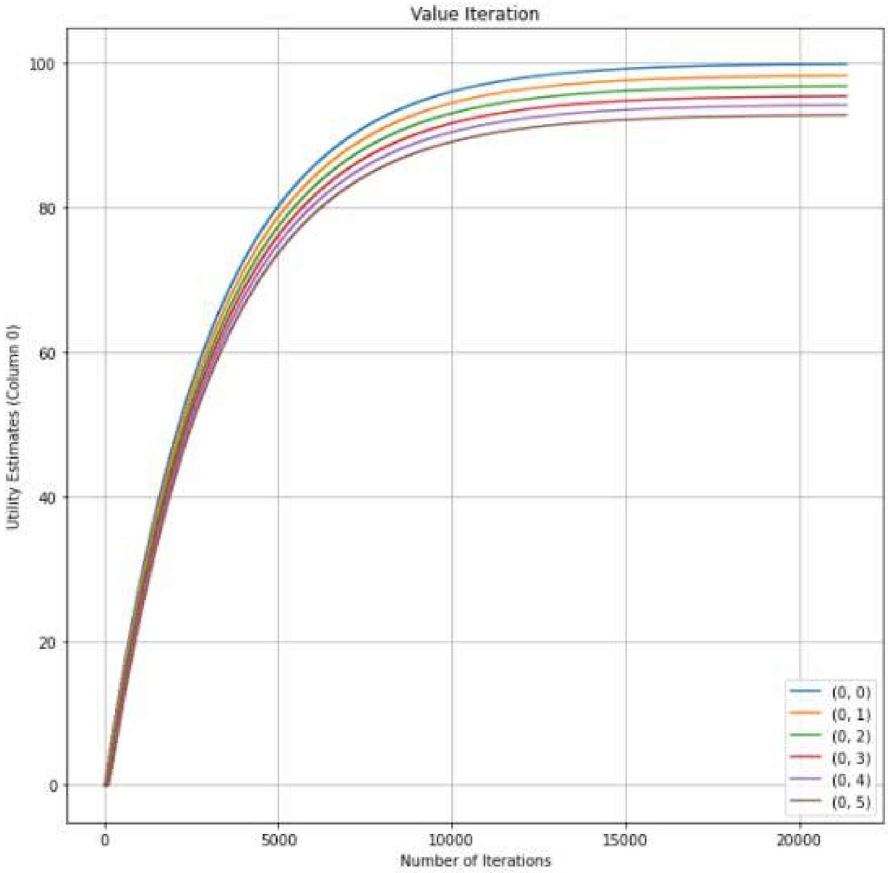
(4,0): 92.55878
(4,2): 93.00696
(4,3): 91.71911
(4,4): 89.45418
(4,5): 88.47489

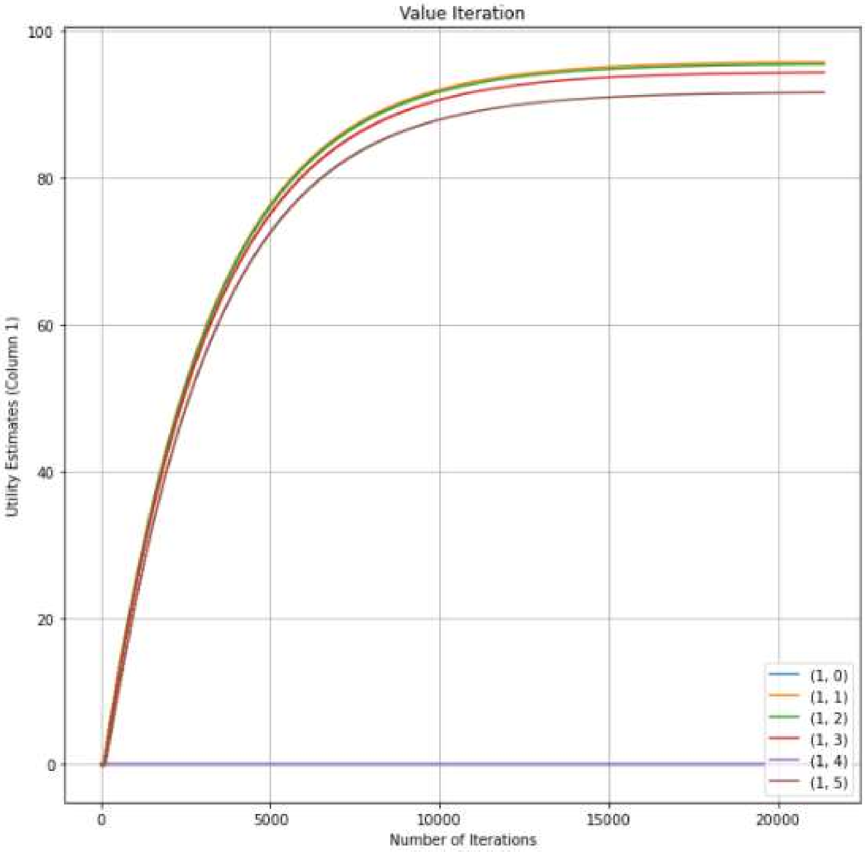
(5,0): 93.23267
(5,1): 90.82305
(5,2): 91.69963
(5,3): 91.79379
(5,4): 90.47248
(5,5): 89.20435

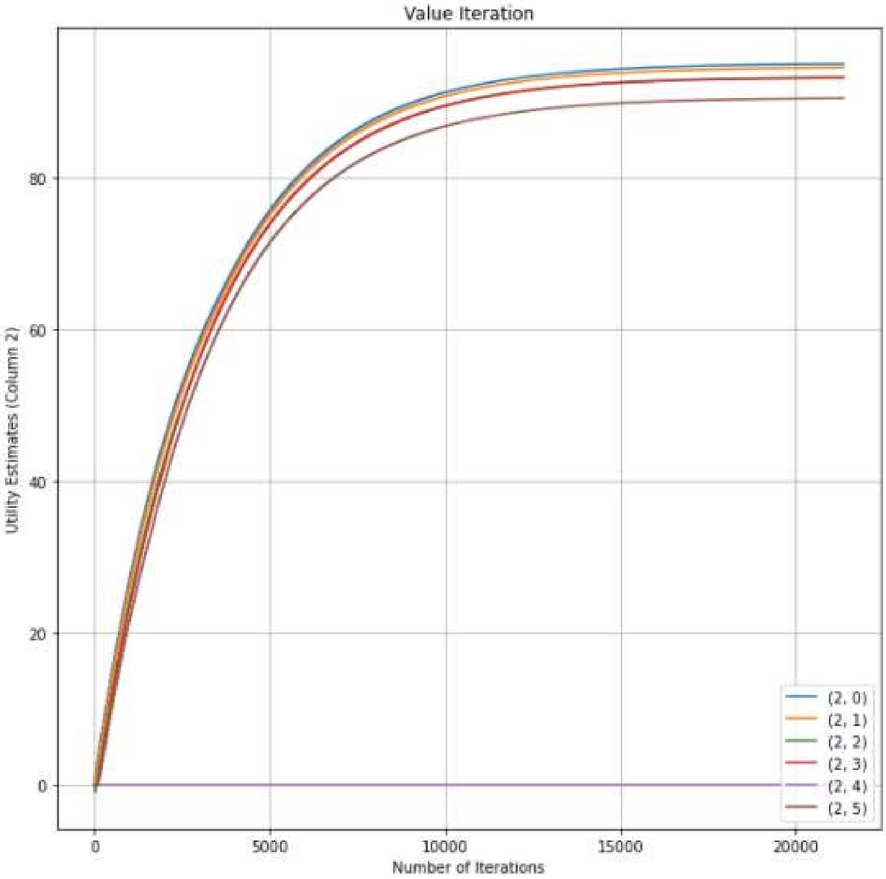
(iv) Plot of utility estimates as a function of the number of iterations

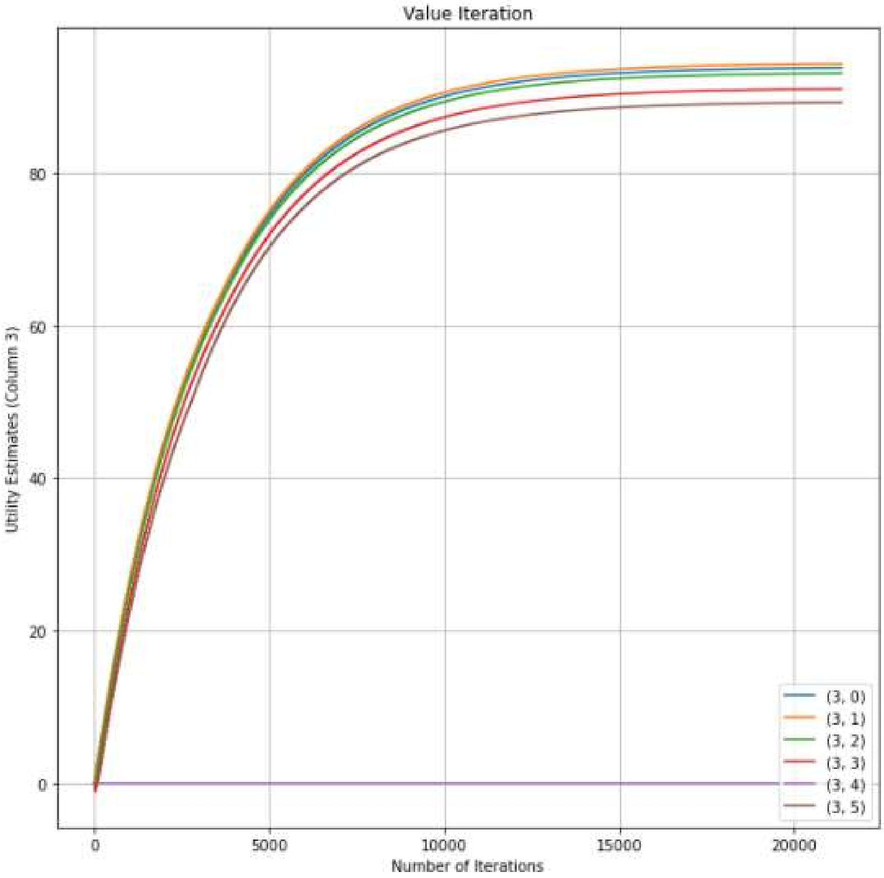


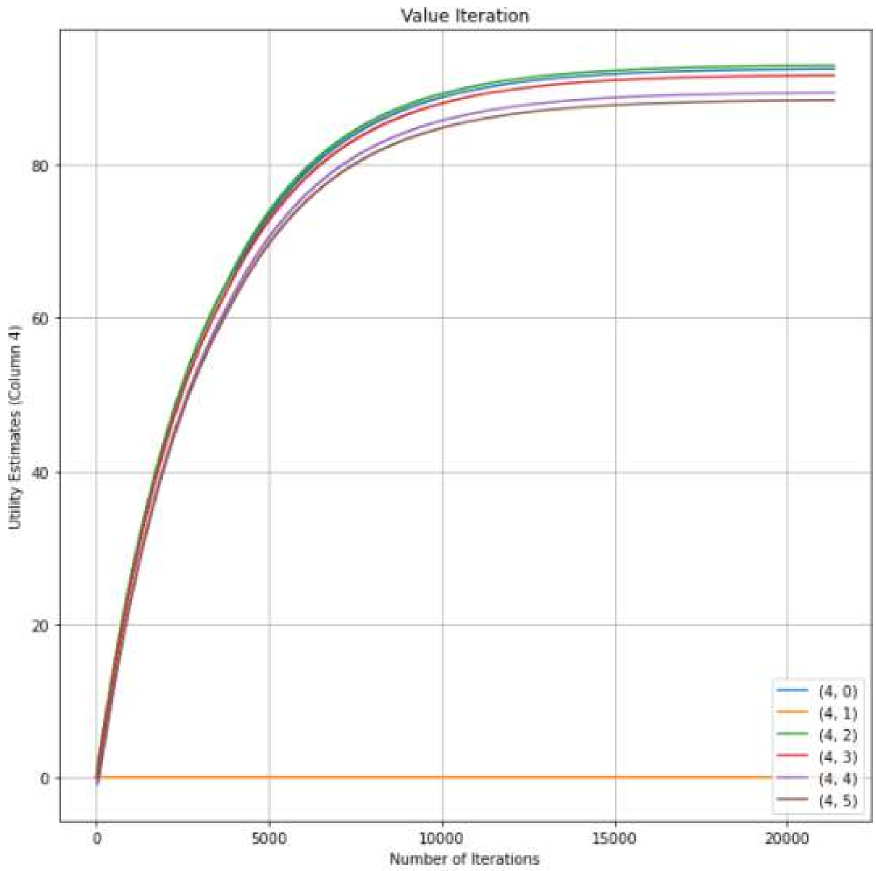
We plot based on the grid columns each cell is in for more clarity:

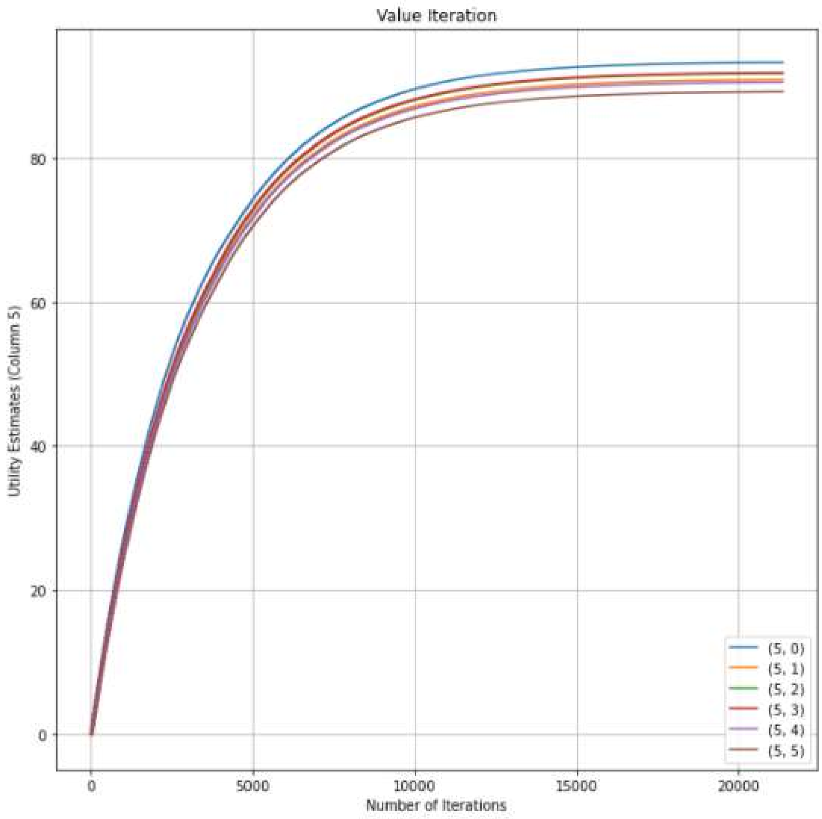





Value Iteration
Utility Estimates (Column 2)
Number of Iterations
0
20
40
60
80
0
5000
10000
15000
20000
(2, 0)
(2, 1)
(2, 2)
(2, 3)
(2, 4)
(2, 5)


Value Iteration
Utility Estimates (Column 3)
Number of Iterations
0
20
40
60
80
0
5000
10000
15000
20000
(3, 0)
(3, 1)
(3, 2)
(3, 3)
(3, 4)
(3, 5)


Value Iteration
Utility Estimates (Column 4)
Number of Iterations
0
20
40
60
80
0
5000
10000
15000
20000
(4, 0)
(4, 1)
(4, 2)
(4, 3)
(4, 4)
(4, 5)


Value Iteration
Utility Estimates (Column 5)
Number of Iterations
0
20
40
60
80
0
5000
10000
15000
20000
(5, 0)
(5, 1)
(5, 2)
(5, 3)
(5, 4)
(5, 5)

Section 2: Policy Iteration

(i) Descriptions of implemented solutions

- State space S
  Define the state space of all possible configurations of the game such as the size of the grid world.

- Action space A
  For each non-wall state, list possible legal actions (U, R, D, L) that bring the agent to the next state.

- Transition function $P(s' \mid s, \pi_i(s))$
  Define the probability of being in state s, taking action a, and ending up in state s'.
  Set if-else conditions such that the agent has a 0.8 probability of going in the intended direction, 0.1 probability of going in a direction 90 degrees clockwise of the intended direction, and 0.1 probability of going in a direction 90 degree anticlockwise of the intended direction.
  For example, if the agent's intended direction is R, there is a 0.8 probability of going R, 0.1 probability of going 90 degrees clockwise which is D, and a 0.1 probability of going 90 degrees anticlockwise which is U.

  Define that if the agent tries to do an action that is not in the list of possible defined legal actions (hits a wall or goes out of the boundary of the state space) for the state, the agent remains in the same state.

- Reward function
  Construct a grid that has the specified rewards at different states.

- Utilities of a state using the Bellman equation

  $$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) U(s')$$

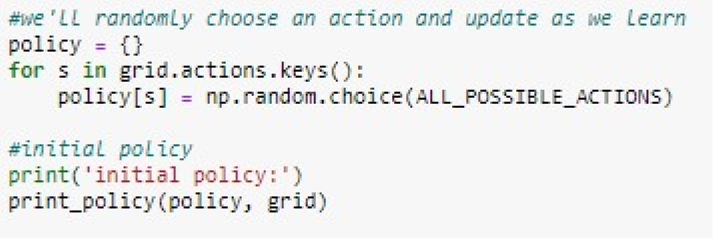
  Define that the utility of a state is the immediate reward for that state plus the expected discounted utility of the next state, assuming that the agent chooses the optimal action.

- Define helper functions that print utility values and that print the policy.

- Set the discount factor gamma to 0.99.

- Initialise all utility values to 0.

- Randomly initialise the policy for each state



```
#we'll randomly choose an action and update as we learn
policy = {}
for s in grid.actions.keys():
    policy[s] = np.random.choice(ALL_POSSIBLE_ACTIONS)

#initial policy
print('initial policy:')
print_policy(policy, grid)
```

- Perform the following policy iteration algorithm:
  Alternate between:
    - Policy evaluation step
      Given the current policy for that iteration step, calculate the utility of each state if the current policy is executed

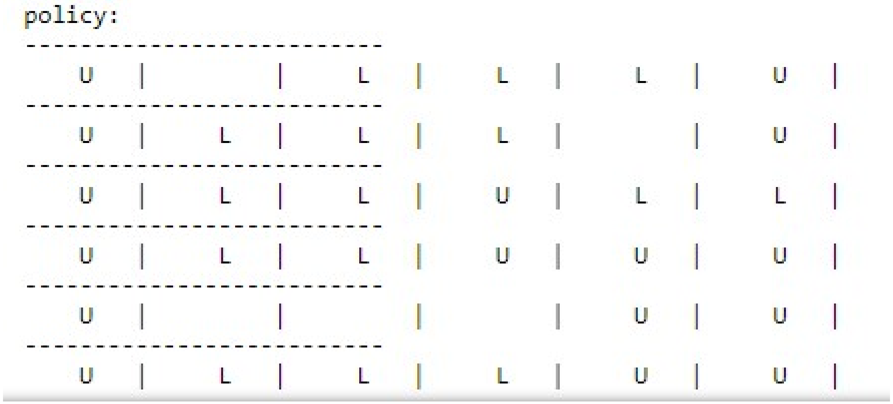
$$U_{i+1}(s) \leftarrow R(s) + \gamma \sum_{s'} P(s' \mid s, \pi_i(s)) U_i(s')$$

    - Policy improvement step
      Given the current utility for that iteration step, find the best policy.

- Break out of the loop if:
    - For an iteration, if the biggest change in utility values for a state is less than 0.0001
    - For an iteration, as compared to the previous iteration, the policy does not change

- Save the history of all values in a list.
  After the utility values have converged, split the saved history of values for each state separately.
  Plot the graphs of Utility Estimation against Number of Iterations for each state onto a single plot.
  Given that there are 36 multiple graphs for each of the 36 states, further plot 6 more plots, with each plot showing the graphs for states in the same column.

(ii) Plot of optimal policy



```
policy:
--------------------------
   U  |      |   L  |   L  |   L  |   U  |
--------------------------
   U  |   L  |   L  |   L  |      |   U  |
--------------------------
   U  |   L  |   L  |   U  |   L  |   L  |
--------------------------
   U  |   L  |   L  |   U  |   U  |   U  |
--------------------------
   U  |      |      |      |   U  |   U  |
--------------------------
   U  |   L  |   L  |   L  |   U  |   U  |
```

Policy at (0,0): Up
Policy at (0,1): Up
Policy at (0,2): Up
Policy at (0,3): Up
Policy at (0,4): Up
Policy at (0,5): Up

Policy at (1,1): Left
Policy at (1,2): Left
Policy at (1,3): Left
Policy at (1,5): Left

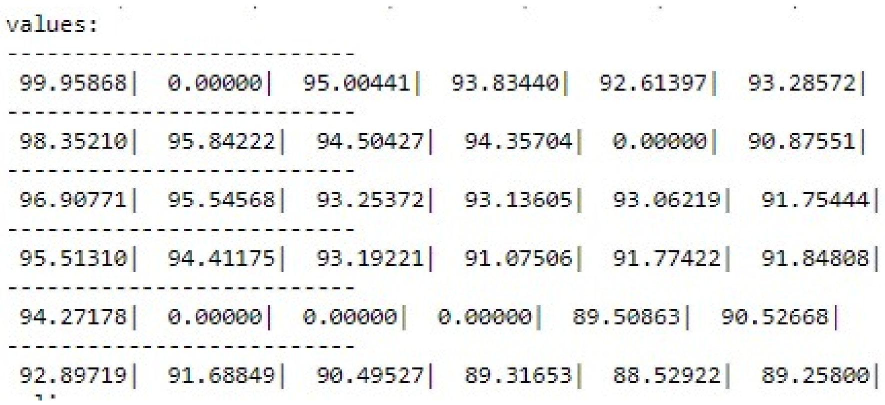
Policy at (2,0): Left
Policy at (2,1): Left
Policy at (2,2): Left
Policy at (2,3): Left
Policy at (2,5): Left

Policy at (3,0): Left
Policy at (3,1): Left
Policy at (3,2): Up
Policy at (3,3): Up
Policy at (3,5): Left

Policy at (4,0): Left
Policy at (4,2): Left
Policy at (4,3): Up
Policy at (4,4): Up
Policy at (4,5): Up

Policy at (5,0): Up
Policy at (5,1): Up
Policy at (5,2): Left
Policy at (5,3): Up
Policy at (5,4): Up
Policy at (5,5): Up

(iii) **Utilities of all states**

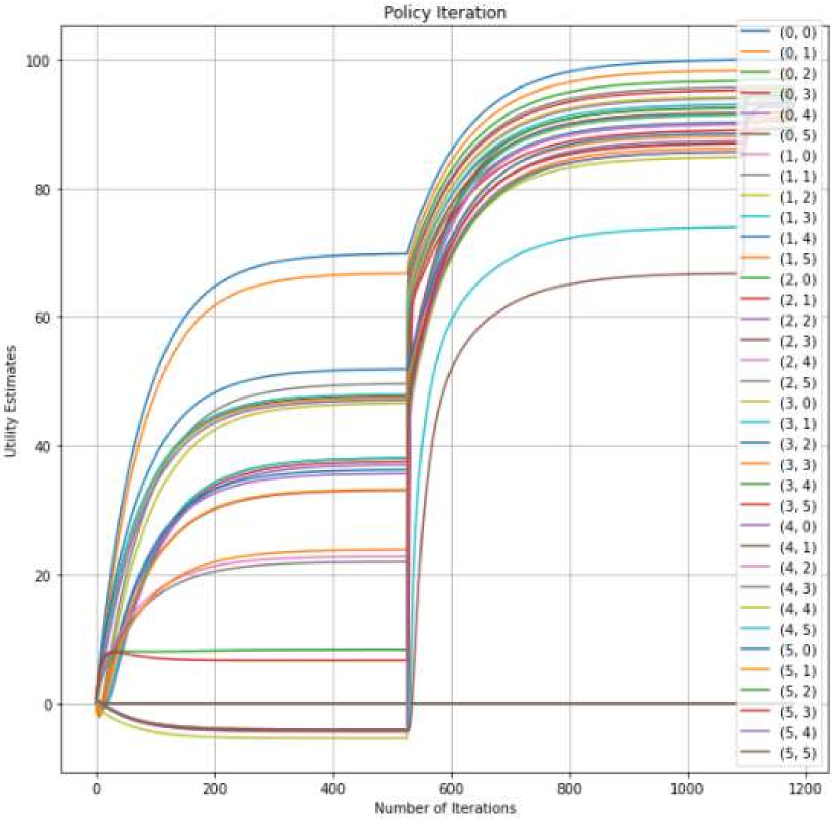


```
values:
--------------------------
 99.95868|  0.00000|  95.00441|  93.83440|  92.61397|  93.28572|
--------------------------
 98.35210|  95.84222|  94.50427|  94.35704|  0.00000|  90.87551|
--------------------------
 96.90771|  95.54568|  93.25372|  93.13605|  93.06219|  91.75444|
--------------------------
 95.51310|  94.41175|  93.19221|  91.07506|  91.77422|  91.84808|
--------------------------
 94.27178|  0.00000|  0.00000|  0.00000|  89.50863|  90.52668|
--------------------------
 92.89719|  91.68849|  90.49527|  89.31653|  88.52922|  89.25800|
```

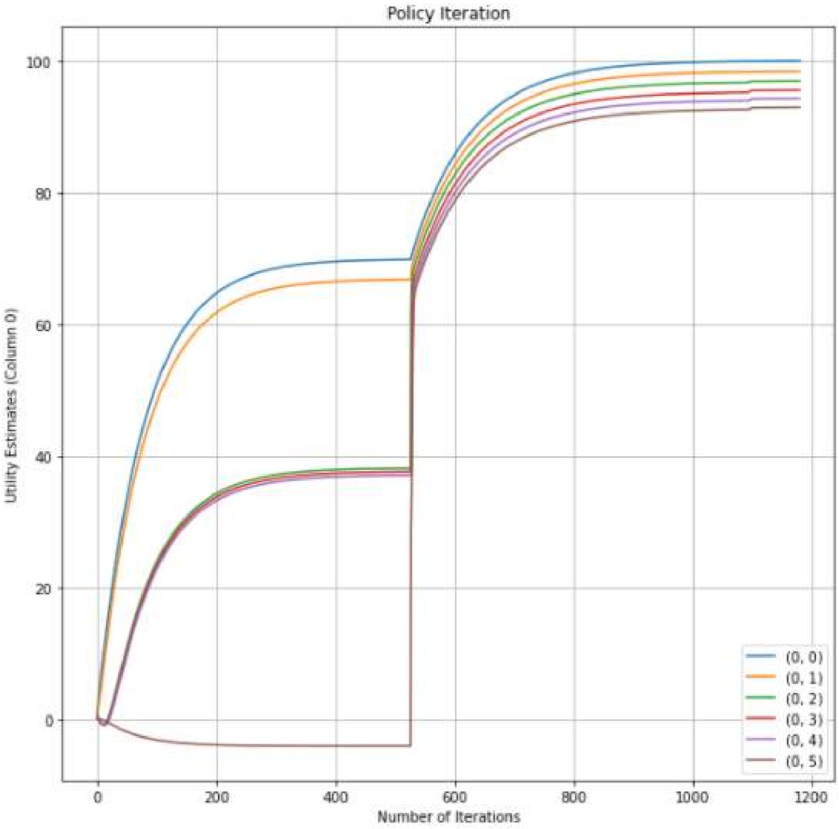
Reference utilities of states:

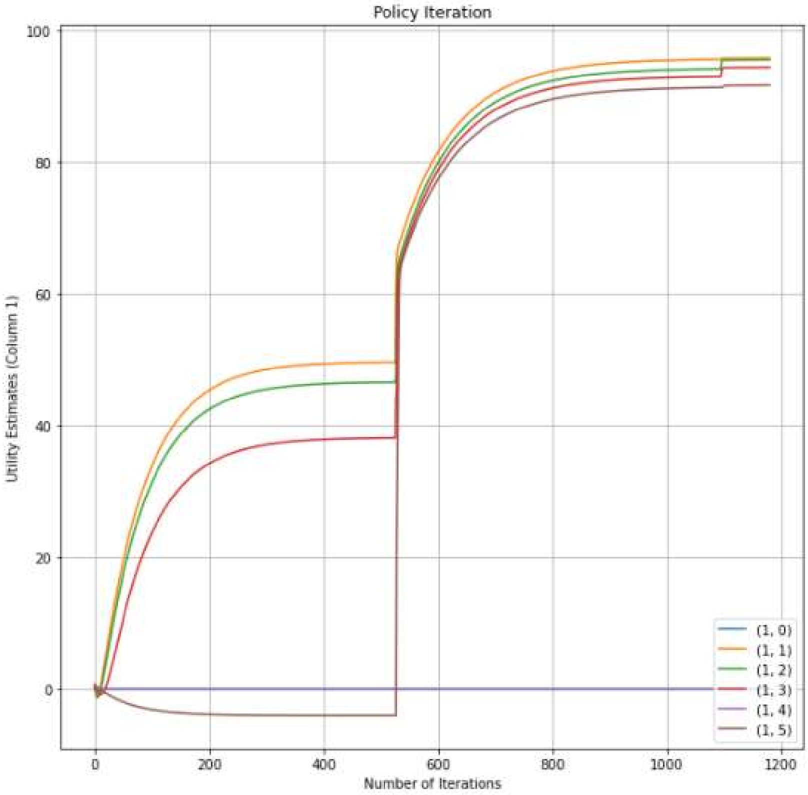
(0,0):	99.95868
(0,1):	98.35210
(0,2):	96.90771
(0,3):	95.51310
(0,4):	94.27178
(0,5):	92.89719
(1,1):	95.84222
(1,2):	95.54568
(1,3):	94.41175
(1,5):	91.68849
(2,0):	95.00441
(2,1):	94.50427
(2,2):	93.25372
(2,3):	93.19221
(2,5):	90.49527
(3,0):	93.83440
(3,1):	94.35704
(3,2):	93.13605
(3,3):	91.07506
(3,5):	89.31653
(4,0):	92.61397
(4,2):	93.06219
(4,3):	91.77422
(4,4):	89.50863
(4,5):	88.52922
(5,0):	93.28572
(5,1):	90.87551
(5,2):	91.75444
(5,3):	91.84808
(5,4):	90.52668
(5,5):	89.25800

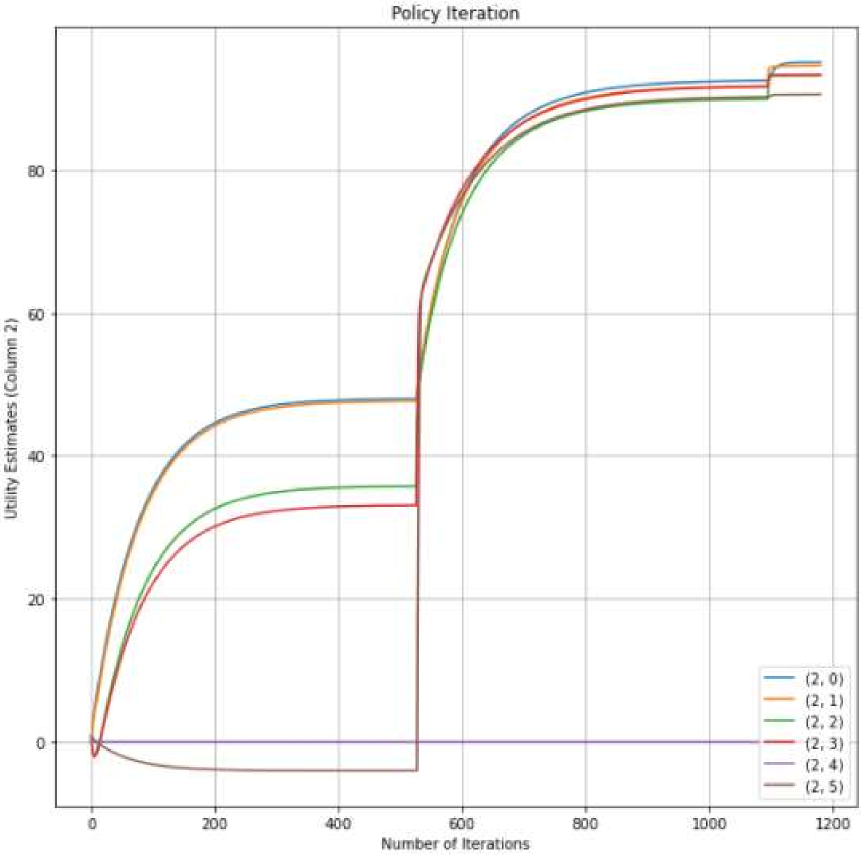
(iv) Plot of utility estimates as a function of the number of iterations

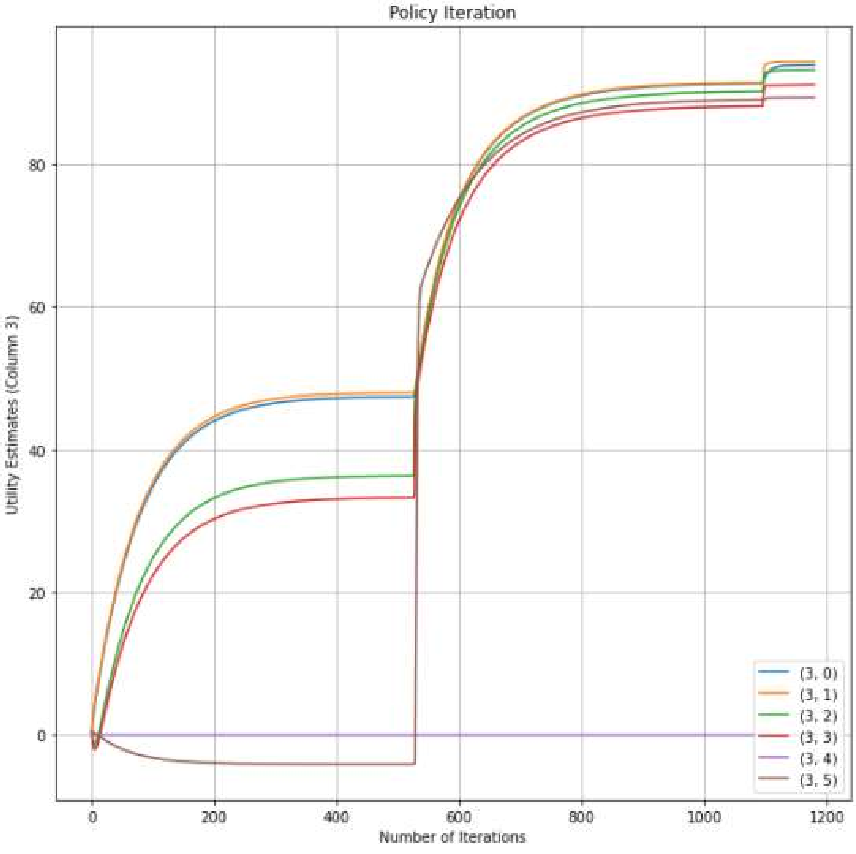


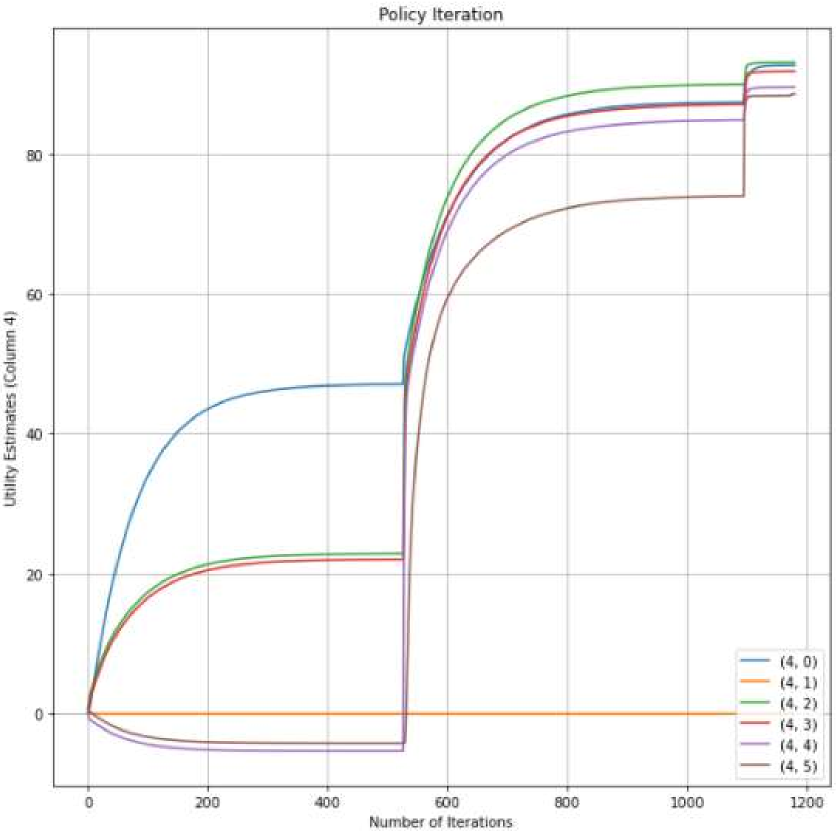
We plot based on the grid columns each cell is in for more clarity:

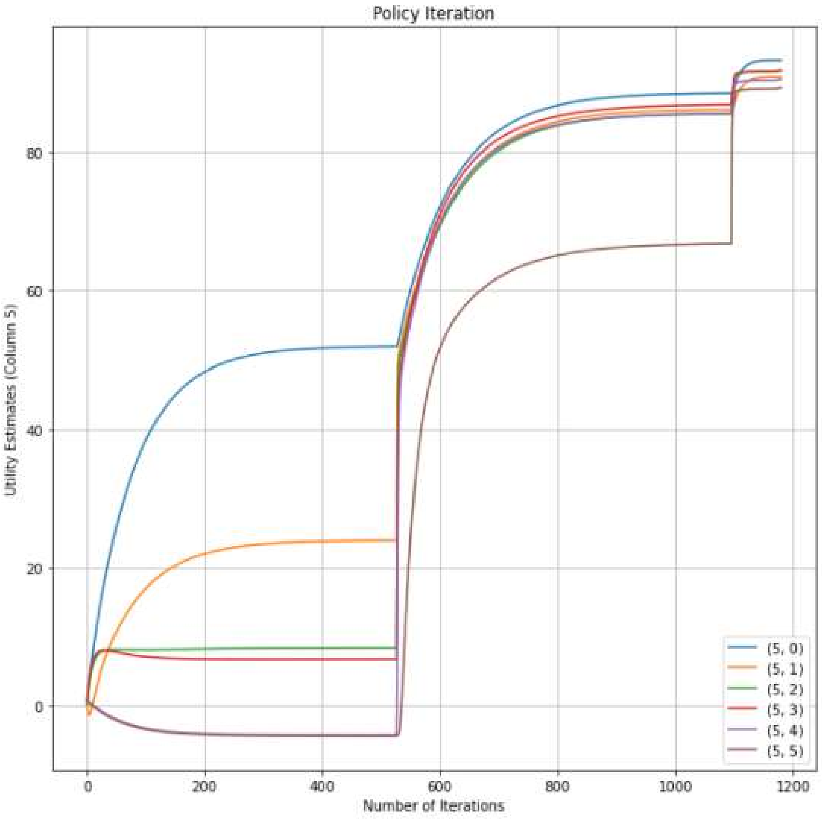





Policy Iteration
Utility Estimates (Column 2)
Number of Iterations
0
20
40
60
80
0
200
400
600
800
1000
1200
(2, 0)
(2, 1)
(2, 2)
(2, 3)
(2, 4)
(2, 5)


Policy Iteration
Utility Estimates (Column 3)
Number of Iterations
0
20
40
60
80
0
200
400
600
800
1000
1200
(3, 0)
(3, 1)
(3, 2)
(3, 3)
(3, 4)
(3, 5)

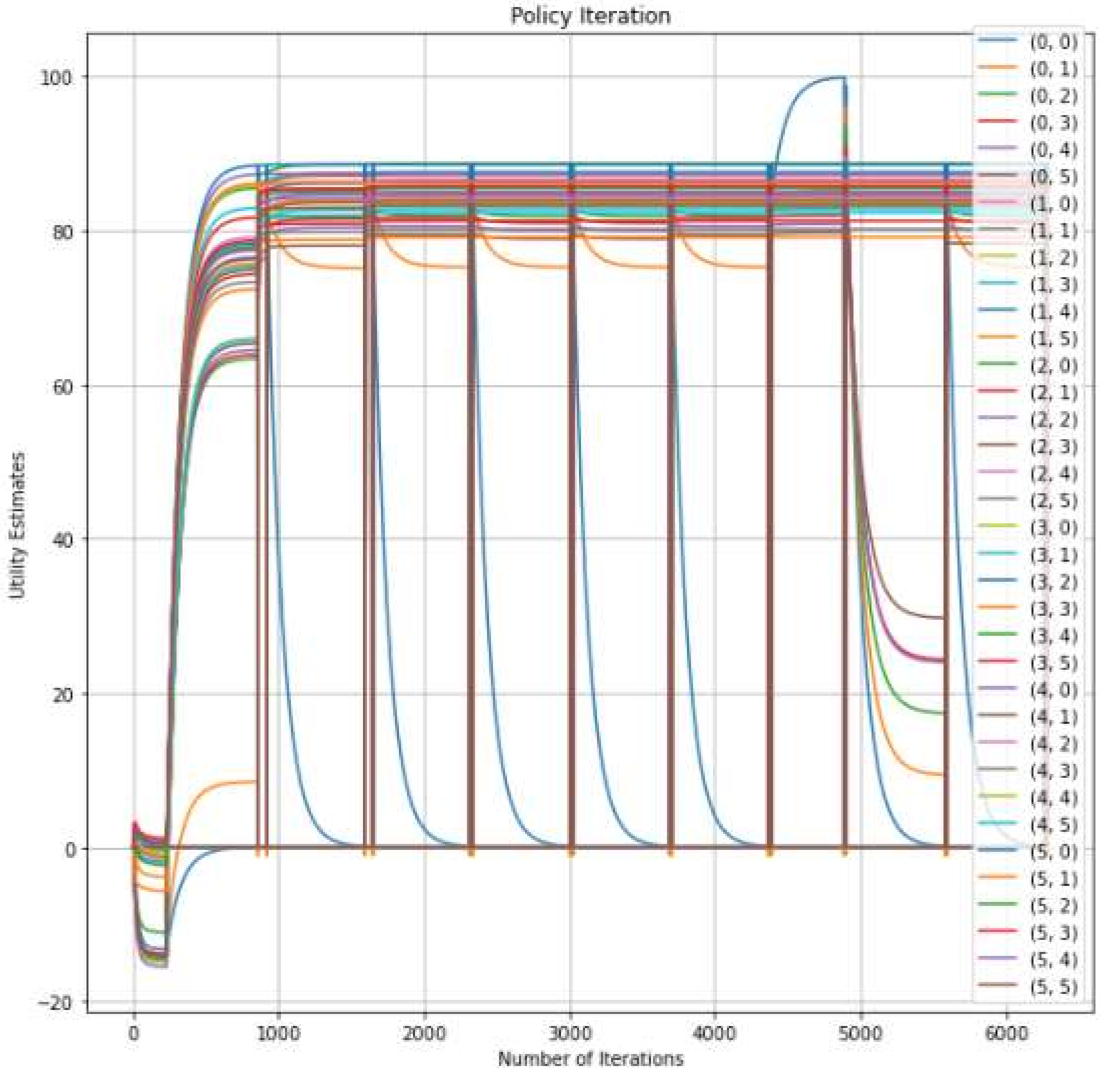

Policy Iteration
Utility Estimates (Column 4)
Number of Iterations
0
20
40
60
80
0
200
400
600
800
1000
1200
(4, 0)
(4, 1)
(4, 2)
(4, 3)
(4, 4)
(4, 5)


Policy Iteration
Utility Estimates (Column 5)
Number of Iterations
0
20
40
60
80
0
200
400
600
800
1000
1200
(5, 0)
(5, 1)
(5, 2)
(5, 3)
(5, 4)
(5, 5)

Part 2

The reward at (0,0) is changed to be 0 or 1 randomly at each iteration (for example one round of policy evaluation and policy improvement). Changing a single cell to have random rewards itself causes swings in the utility estimates of the cells surrounding it and this effect is propagated to other cells. The number of iterations before our defined algorithm termination condition is met increases multi-fold. For example for policy iteration, the number of iterations increased approximately three-fold from 1181 to 6290.

If there are more cells that are changed to have random rewards at each utilisation, then we may expect wilder swings in the utility estimates, and it will take more iterations and a longer time to reach convergence.



As the number of states or the complexity of the environment increases, it will take more time to reach convergence.