NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

CE/CZ4046 INTELLGENT AGENTS

Assignment 1

Lim Song Wei, Greg   U2120517G

Contents

Set Up and Helper Functions

Maze Implementation

The maze can be represented with 2 components, the “abstract” maze, containing walls and layout, and the rewards. The maze is generated with a default empty cell reward of -0.04.

```
def makeGridReward(width, height, plusOne, minusOne, wall):
    grid = np.zeros((height, width))
    reward = np.full((height, width),-0.04)
    U = grid.copy()

    for i in plusOne:
        grid[i] = 1
        reward[i] = 1
    for i in minusOne:
        grid[i] = -1
        reward[i] = -1
    for i in wall:
        grid[i[0]][i[1]] = np.nan

    return (grid,reward)
```

Code 1.1: Function to generate Grid and Reward Matrix

```
def getRelVal(x,y,U,dir,walls):
    """
    Return the expected utilily for a certain direction relertive to a cell
    if the relertive direction is a wall, return its own utility
    """
    height = len(U)
    width = len(U[0])
    def isValid(x,y):
        if(x<0 or x>=height): return False
        if(y<0 or y>=width): return False
        if (x,y) in walls: return False
        return True

    if dir == "^":
        # x,y = x,y-1
        return U[x-1][y] if isValid(x-1,y) else U[x][y]
    elif dir == ">":
        # x,y = x+1,y
        return U[x][y+1] if isValid(x,y+1) else U[x][y]
    elif dir == "v":
        # x,y = x,y+1
        return U[x+1][y] if isValid(x+1,y) else U[x][y]
    elif dir == "<":
        # x,y = x-1,y
        return U[x][y-1] if isValid(x,y-1) else U[x][y]
```

Code 1.2: helper function to return an expected value given a direction with 100% chance

Implementation of Value and Policy Iterations

Value Iteration

For every possible state, its value is updated on the best possible move it could make based on the previous utilities. To execute one step of value iteration,

1. Loop through all states in the grid
2. Calculate expected value for each direction for this state.
3. Update this state to the highest value.

```
def iterateValue(U, reward, grid, walls):
    '''
    performs one step of value iteration, returns the U after 1 step.
    '''
    height = len(grid)
    width = len(grid[0])
    Ui = np.zeros((height, width))

    t = [0,0,0,0]

    for h in range(height):
        for w in range(width):
            if (h,w) in walls:
                Ui[h][w] = 0
                continue
            t[0] = reward[h][w] + GAMMA*(getRelVal(h,w,U,"<",walls)*0.1 +
getRelVal(h,w,U,"^",walls)*0.8 + getRelVal(h,w,U,">",walls)*0.1)
            t[1] = reward[h][w] + GAMMA*(getRelVal(h,w,U,"^",walls)*0.1 +
getRelVal(h,w,U,">",walls)*0.8 + getRelVal(h,w,U,"v",walls)*0.1)
            t[2] = reward[h][w] + GAMMA*(getRelVal(h,w,U,">",walls)*0.1 +
getRelVal(h,w,U,"v",walls)*0.8 + getRelVal(h,w,U,"<",walls)*0.1)
            t[3] = reward[h][w] + GAMMA*(getRelVal(h,w,U,"v",walls)*0.1 +
getRelVal(h,w,U,"<",walls)*0.8 + getRelVal(h,w,U,"^",walls)*0.1)
            Ui[h][w] = np.max(t)
    return Ui
```

Code 2.1: One Step Value Iterate Function

Policy Iteration

To perform policy iteration, 2 functions are needed, policy evaluation and policy improvement.

For policy evaluation, we keep updating the utility until it converges using the policy it is given. To execute Policy Evaluation,

1. Loop till delta between current and previous utility is small enough.
2. Calculate expected value for each direction for this state given only the policy given
3. Update utility

```
def evaluatePolicy(p, U2, reward, grid, walls):
    '''
    evaluates a given policy and return the expected U of this Policy
    '''
    height = len(grid)
    width = len(grid[0])
    # U = np.zeros((height, width))
    deltaChange=1

    U = U2.copy()

    # while delta > THETA:
    # for i in range(1000):
    while deltaChange>1e-5:
        for h in range(height):
            for w in range(width):
                if (h,w) in walls: continue

                oldU = U.copy()

                if p[h][w] == 0: U[h][w] = reward[h][w] + GAMMA*(getRelVal(h,w,U,"<",walls)*0.1 + 
getRelVal(h,w,U,"^",walls)*0.8 + getRelVal(h,w,U,">",walls)*0.1)
                if p[h][w] == 1: U[h][w] = reward[h][w] + GAMMA*(getRelVal(h,w,U,"^",walls)*0.1 + 
getRelVal(h,w,U,">",walls)*0.8 + getRelVal(h,w,U,"v",walls)*0.1)
                if p[h][w] == 2: U[h][w] = reward[h][w] + GAMMA*(getRelVal(h,w,U,">",walls)*0.1 + 
getRelVal(h,w,U,"v",walls)*0.8 + getRelVal(h,w,U,"<",walls)*0.1)
                if p[h][w] == 3: U[h][w] = reward[h][w] + GAMMA*(getRelVal(h,w,U,"v",walls)*0.1 + 
getRelVal(h,w,U,"<",walls)*0.8 + getRelVal(h,w,U,"^",walls)*0.1)

                deltaChange = np.average(np.absolute(oldU - U))

    return U
```

Code 2.2 a: One Step Policy Evaluation Function

For policy improvement, we use the given utility function and iterate over all states to identify the best policy to get the best utility. To execute Policy Improvement,

1. Loop through all states
2. Calculated the expected utility for taking all possible actions
3. Update the policy of each cell to the best policy for the given utility

```
def improvePolicy(U, reward, grid, walls):
        '''
    Returns a policy for a given U
    Note: 0 is up 1 is right 2 is down and 3 is right
    '''
    height = len(grid)
    width = len(grid[0])
    P = np.zeros((height, width))

    t = [0,0,0,0]

    for h in range(height):
        for w in range(width):
            if (h,w) in walls: continue

            t[0] = reward[h][w] + getRelVal(h,w,U,"<",walls)*0.1 + getRelVal(h,w,U,"^",walls)*0.8 +
getRelVal(h,w,U,">",walls)*0.1
            t[1] = reward[h][w] + getRelVal(h,w,U,"^",walls)*0.1 + getRelVal(h,w,U,">",walls)*0.8 +
getRelVal(h,w,U,"v",walls)*0.1
            t[2] = reward[h][w] + getRelVal(h,w,U,">",walls)*0.1 + getRelVal(h,w,U,"v",walls)*0.8 +
getRelVal(h,w,U,"<",walls)*0.1
            t[3] = reward[h][w] + getRelVal(h,w,U,"v",walls)*0.1 + getRelVal(h,w,U,"<",walls)*0.8 +
getRelVal(h,w,U,"^",walls)*0.1

            P[h][w] = np.argmax(t)
    return P
```

Code 2.2b: One Step Policy Improvement Function

Part 1

Making the Gird

```
WIDTH = 6
HEIGHT = 6
START = (3,2)
WALLS = ((0,1), (1,4), (4,1), (4,2), (4,3),)
PLUSONE = ((0,0), (0,2), (0,5), (1,3), (2,4), (3,5),)
MINUSONE = ((1,1), (1,5), (2,2), (3,3), (4,4),)

(grid,reward) = makeGridReward(WIDTH, HEIGHT, PLUSONE, MINUSONE, WALLS)

prettyPrint(grid)
```

Code 3.1: Set Up of grid given in the assignment

```
+----+----+----+----+----+----+
| +1 | [] | +1 |    |    | +1 |
+----+----+----+----+----+----+
|    | -1 |    | +1 | [] | -1 |
+----+----+----+----+----+----+
|    |    | -1 |    | +1 |    |
+----+----+----+----+----+----+
|    |    |    | -1 |    | +1 |
+----+----+----+----+----+----+
|    | [] | [] | [] | -1 |    |
+----+----+----+----+----+----+
|    |    |    |    |    |    |
+----+----+----+----+----+----+
```

Figure 3.2: Grid Generated

For Part 1:

Gamma:	0.99
Dimensions:	6x6
Convergence threshold:	10^-5

Value Iteration

To perform value iteration, we would keep updating U with the previous U value and record the average absolute change for all states. It is determined to have converge when delta change is less than 10^-5. To get the policy, we borrow improve Policy function to get a policy from the utility,

```
U = np.zeros((HEIGHT, WIDTH))
oldU = np.zeros((HEIGHT, WIDTH))
deltaChange = 1
deltaHistory = []
allU=[]

while deltaChange >pow(10,-5): # converges when absolute sum of difference is > 10^-5
    oldU = U

    U = iterateValue(U, reward, grid, WALLS)
    deltaChange = np.average(np.absolute(oldU - U))
    deltaHistory.append(float(deltaChange))

    allU.append(U)
    tU = U.copy()
    tU[tU == 0] = np.nan

# borrow improvePolicy function to generate policy
P = improvePolicy(U, reward, grid, WALLS)
```

Code 3.3: Performing value Iteration on small grid

```
 100.00          95.04   93.87   92.65   93.33

  98.39   95.88  94.54   94.40           90.92

  96.95   95.59  93.29   93.18   93.10   91.79

  95.55   94.45  93.23   91.11   91.81   91.89

  94.31                          89.55   90.57

  92.94   91.73  90.53   89.36   88.57   89.30
```

Figure 3.4a: Utility of small grid using value iteration

```
+---+---+---+---+---+---+
| ^ |   | < | < | < | ^ |
+---+---+---+---+---+---+
| ^ | < | < | < |   | ^ |
+---+---+---+---+---+---+
| ^ | < | < | ^ | < | < |
+---+---+---+---+---+---+
| ^ | < | < | ^ | ^ | ^ |
+---+---+---+---+---+---+
| ^ |   |   |   | ^ | ^ |
+---+---+---+---+---+---+
| ^ | < | < | < | ^ | ^ |
+---+---+---+---+---+---+
```

Figure 3.4b: Policy of small grid using value iteration

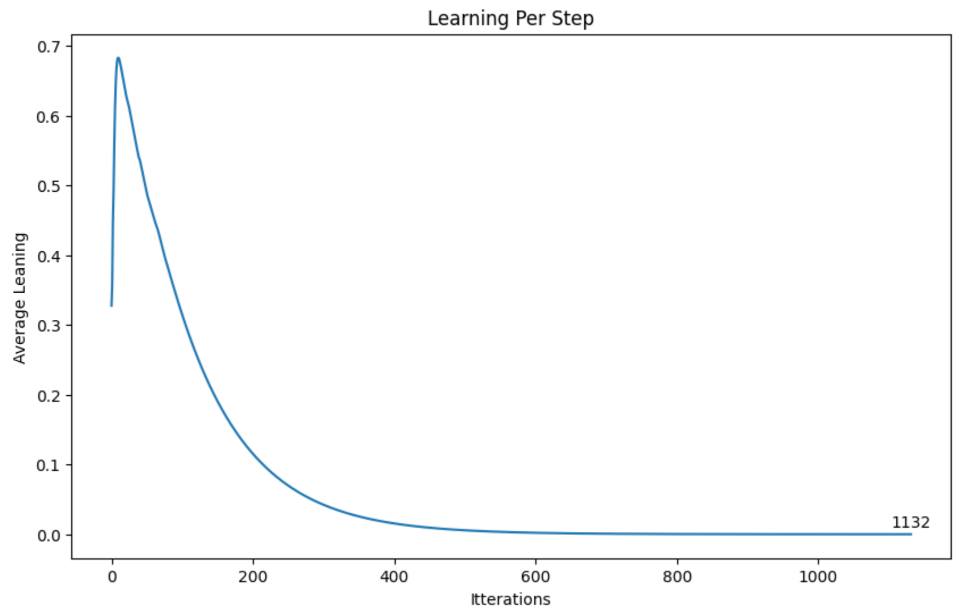


Figure 3.5: Learning rate of small grid using value iteration

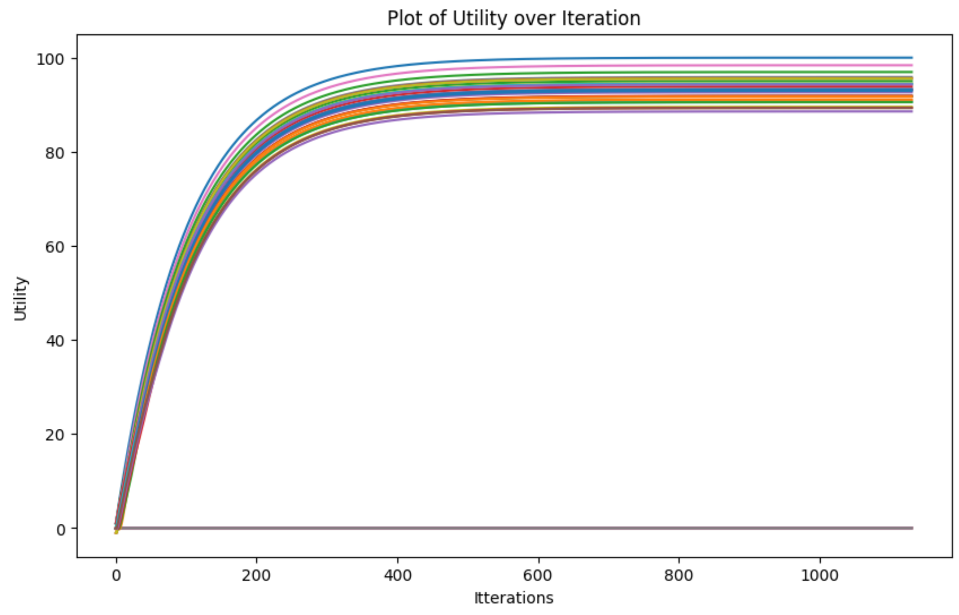


Figure 3.6: Change in utility of small grid using value iteration

From Figure 3.5 and 3.5, it can be shown that the learning rate converges to 0 and the utility of all cells converges to a value. This aligns with the theory that value iteration converges.

Policy Iteration

To perform policy Iteration, we would keep track of the policy from iteration to iteration to check if it is consistent. If it is consistent, the learning has completed. To learn,

1. While old policy does not equal new policy,
2. Evaluate the current policy
3. Pass the evaluated utility to improve the policy

```
U = np.zeros((HIGHT, WIDTH))
P = improvePolicy(U, reward, grid, WALLS)
oldU = np.zeros((HIGHT, WIDTH))
deltaChange = 1
deltaHistory = []
allU=[]

stable = False
while not stable: # converges when absolute sum of difference is > 10^-5
    oldU = U.copy()
    oldP = P.copy()

    U = evaluatePolicy(P, U, reward, grid, WALLS)
    P = improvePolicy(U, reward, grid, WALLS)

    if(np.array_equal(oldP, P)):
        stable = True

    deltaChange = np.average(np.absolute(oldU - U))
    deltaHistory.append(float(deltaChange))

    allU.append(U)
```

Code 3.7: Performing policy Iteration on small grid

```
 100.00          95.04   93.87   92.65   93.30

  98.39   95.88   94.54   94.40           90.89

  96.95   95.59   93.29   93.17   93.10   91.79

  95.55   94.45   93.23   91.11   91.81   91.88

  94.31                           89.55   90.56

  92.94   91.73   90.53   89.36   88.57   89.29
```

Figure 3.8a: Utility of small grid using policy iteration

```
+---+---+---+---+---+---+
| ^ |   | < | < | < | ^ |
+---+---+---+---+---+---+
| ^ | < | < | < |   | ^ |
+---+---+---+---+---+---+
| ^ | < | < | ^ | < | < |
+---+---+---+---+---+---+
| ^ | < | < | ^ | ^ | ^ |
+---+---+---+---+---+---+
| ^ |   |   |   | ^ | ^ |
+---+---+---+---+---+---+
| ^ | < | < | < | ^ | ^ |
+---+---+---+---+---+---+
```

Figure 3.8b: Policy of small grid using policy iteration

From figure 3.4b and 3.8b, both methods have converged on the same policy. Additionally, both have almost identical utility values. They differ a little as the thresh hold is only 10^-5.

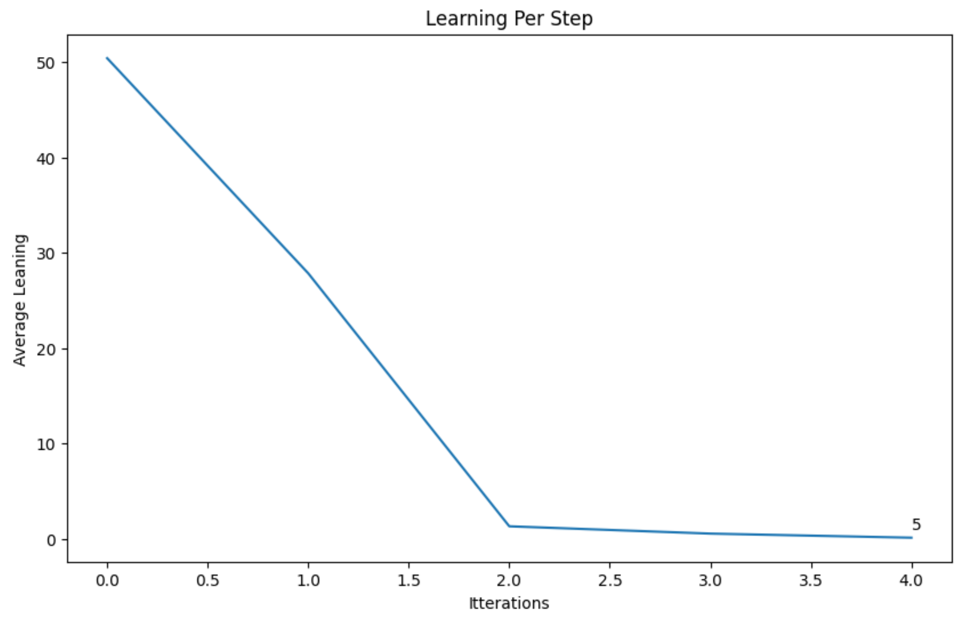


Figure 3.9: Learning rate of small grid using policy iteration

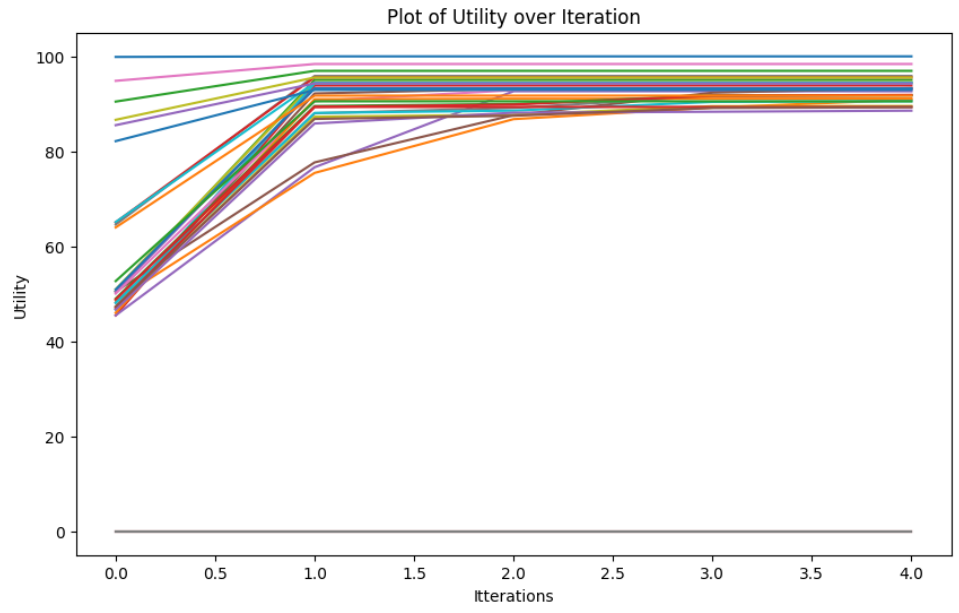


Figure 3.10: Change in utility of small grid using policy iteration

It can be observed from figure 3.5 and figure 3.9 that value iteration took 1132 iterations to converge, and policy iteration took 5 to converge. It is also shown from figure 3.6 and figure 3.10 that each step is policy iteration results in a much greater change in the expected utility.

Part 2 – Larger Maze

Set Up

```
TOTALWALLS = 20
TOTALPLUSONE = 20
TOTALMINUSONE = 20

random.seed("A for SC4003")

def gencoordinates(w, h):
    w=w-1
    h=h-1
    seen = set()
    x, y = random.randint(0, h), random.randint(0, w)
    while True:
        seen.add((x, y))
        yield (x, y)
        x, y = random.randint(0, h), random.randint(0, w)
        while (x, y) in seen:
            x, y = random.randint(0, h), random.randint(0, w)

WIDTH = 16
HEIGHT = 12

WALLS = []
PLUSONE = []
MINUSONE = []

if (TOTALWALLS+TOTALMINUSONE+TOTALPLUSONE)*2> (WIDTH*HEIGHT):
    print("Not enought blank tiles")
else:
    g = gencoordinates(WIDTH, HEIGHT)

    for i in range(TOTALWALLS):
        WALLS.append(next(g))
    for i in range(TOTALPLUSONE):
        PLUSONE.append(next(g))
    for i in range(TOTALMINUSONE):
        MINUSONE.append(next(g))

    (grid,reward) = makeGridReward(WIDTH, HEIGHT, PLUSONE, MINUSONE, WALLS)



    prettyPrint(grid)
    # print()
    # prettyPrint(reward)
```

Figure 3.1: Large maze set up 16x12

```
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
| +1 |    |    |    |    |    |    | +1 |    |    |    |    | +1 |    |    |    |
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
| +1 | +1 |    |    | +1 | -1 |    | -1 |    | [] |    |    |    |    |    |    |
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
|    |    |    | +1 | -1 | -1 |    |    |    |    |    |    |    | -1 |    |    |
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
| [] | +1 |    |    | -1 |    | [] |    |    |    | +1 | +1 | [] |    | -1 |    |
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
|    | -1 |    |    |    | [] | [] |    |    |    |    |    |    |    |    | +1 |
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
|    |    |    |    |    |    |    |    | [] |    | +1 | -1 | +1 | -1 |    | [] |
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
|    |    |    |    |    |    | -1 | [] | [] |    |    |    |    | -1 | +1 | +1 |
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
| [] | +1 |    |    |    | -1 |    | -1 |    |    |    |    |    | -1 |    |    |
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
|    |    |    |    |    |    | [] |    |    | [] |    | -1 |    |    |    |    |
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
|    |    |    |    |    |    | [] |    |    |    |    | -1 | -1 |    | [] |    |
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
|    | -1 |    |    | [] | [] | +1 |    |    |    | +1 |    |    |    |    |    |
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
|    | +1 |    | +1 | [] |    | [] |    | [] |    |    |    |    |    |    | -1 |
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
```
Figure 3.2: Large maze grid

For Part 2:

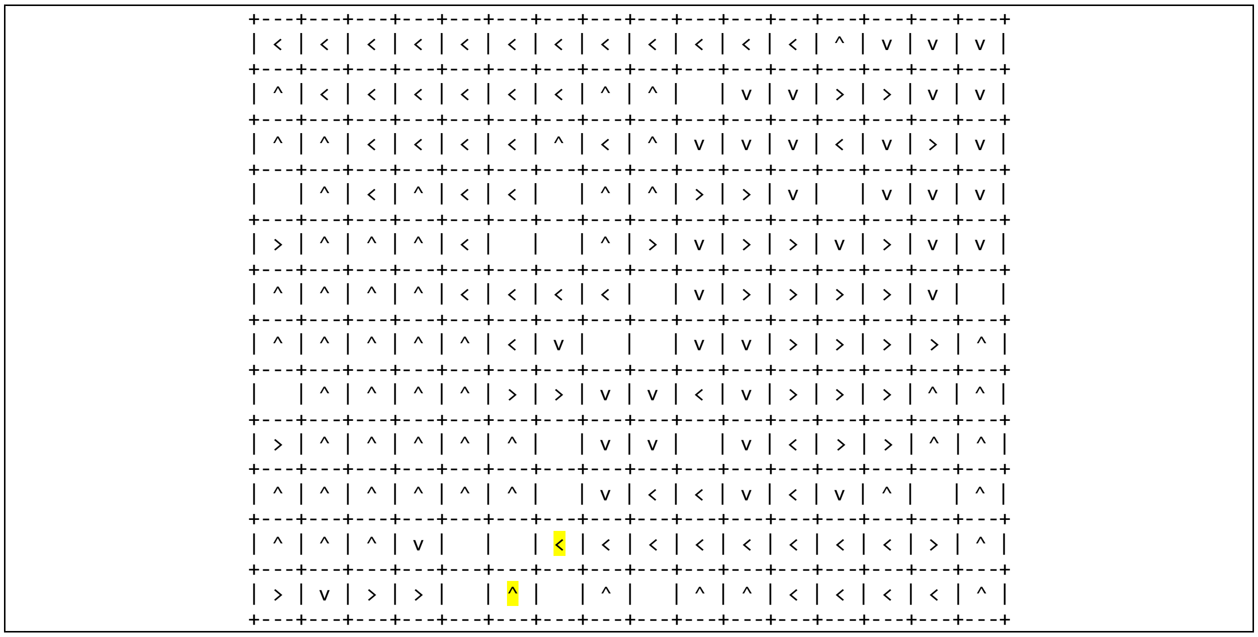
Gamma: 0.99

Dimensions: 16x12

Convergence threshold: 10^-5

Value Iteration on Larger Maze

```
99.70  98.51  97.31  96.10  95.01  93.75  92.51  92.33  90.99  89.80  88.52  87.39  87.76  86.89  87.82  88.75

99.67  99.39  97.90  96.58  96.23  93.61  92.27  90.15  89.84         87.58  87.58  86.86  87.81  88.97  90.04

98.33  98.00  96.82  96.67  94.29  92.04  91.03  89.74  88.69  87.76  88.73  88.82  87.58  88.69  90.16  91.36

       97.85  96.46  95.28  92.97  91.79         88.48  87.74  88.68  90.02  90.16         91.03  91.26  92.71

93.83  95.20  95.08  94.01  92.68                87.31  87.16  88.24  89.12  90.06  91.22  92.20  93.52  94.12

     92.74  93.82  93.74  92.76  91.55  90.23  89.06  87.84         89.43  90.12  90.14  92.45  92.52  94.85

91.64  92.49  92.42  91.52  90.36  89.27  89.09                90.53  90.02  90.96  92.34  93.66  96.38  96.72

       92.43  91.25  90.31  89.28  88.97  91.46  92.96  93.15  91.78  91.03  90.28  91.42  92.57  94.90  95.40

89.65  90.96  90.06  89.14  88.19  87.85         95.56  94.57         92.22  90.03  90.96  92.23  93.57  94.09

88.60  89.57  88.86  87.98  87.10  86.75         96.94  95.69  94.56  93.71  91.39  89.91  90.91         92.87

87.42  87.27  87.64  87.54               100.00  98.37  96.94  95.42  95.13  93.53  92.00  90.77  90.20  91.50

86.68  87.73  87.54  88.68         -4.00         97.09         94.13  93.80  92.69  91.54  90.39  89.32  89.14
```
Figure 3.3: Utility of large grid using value iteration



```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| < | < | < | < | < | < | < | < | < | < | < | < | ^ | v | v | v |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| ^ | < | < | < | < | < | < | ^ | ^ |   | v | v | > | > | v | v |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| ^ | ^ | < | < | < | < | ^ | < | ^ | v | v | v | < | v | > | v |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|   | ^ | < | ^ | < | < |   | ^ | ^ | > | > | v |   | v | v | v |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| > | ^ | ^ | ^ | < |   |   | ^ | > | v | > | > | v | > | v | v |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| ^ | ^ | ^ | ^ | < | < | < | < |   | v | > | > | > | > | v |   |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| ^ | ^ | ^ | ^ | ^ | < | v |   |   | v | v | > | > | > | > | ^ |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|   | ^ | ^ | ^ | ^ | > | > | v | v | < | v | > | > | > | ^ | ^ |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| > | ^ | ^ | ^ | ^ | ^ |   | v | v |   | v | < | > | > | ^ | ^ |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| ^ | ^ | ^ | ^ | ^ | ^ |   | v | < | < | v | < | v | ^ |   | ^ |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| ^ | ^ | ^ | v |   |   | < | < | < | < | < | < | < | < | > | ^ |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| > | v | > | > |   | ^ |   | ^ |   | ^ | ^ | < | < | < | < | ^ |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

Figure 3.4: Policy of large grid using value iteration

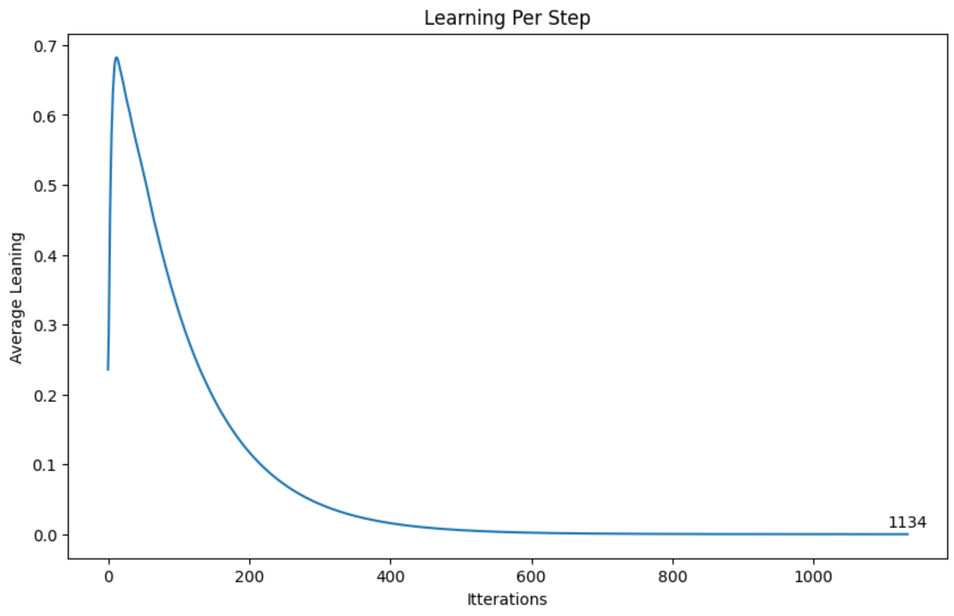


Figure 3.5: Learning rate of large grid using value iteration

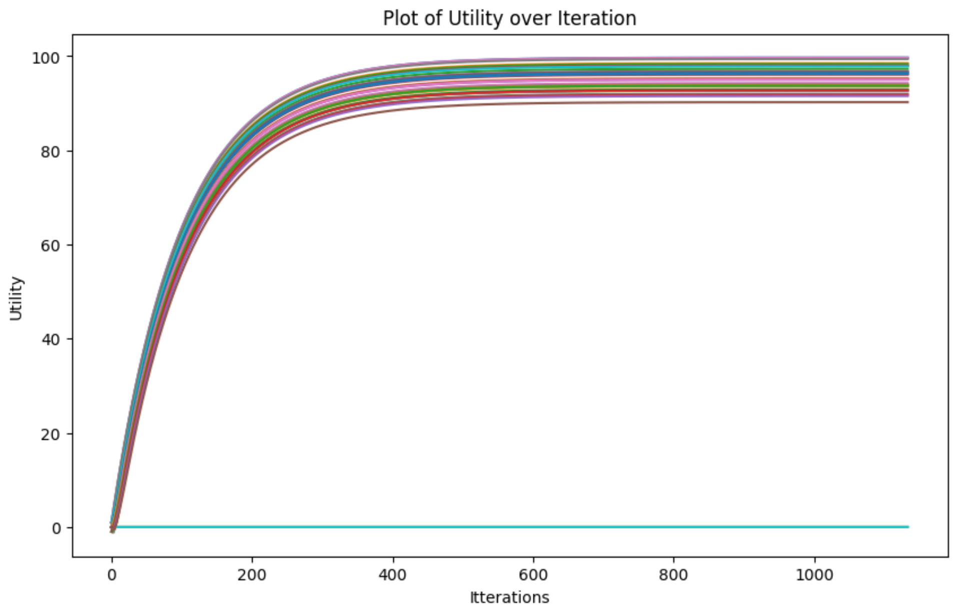


Figure 3.6: Change in utility of large grid using value iteration

Policy Iteration on Larger Maze

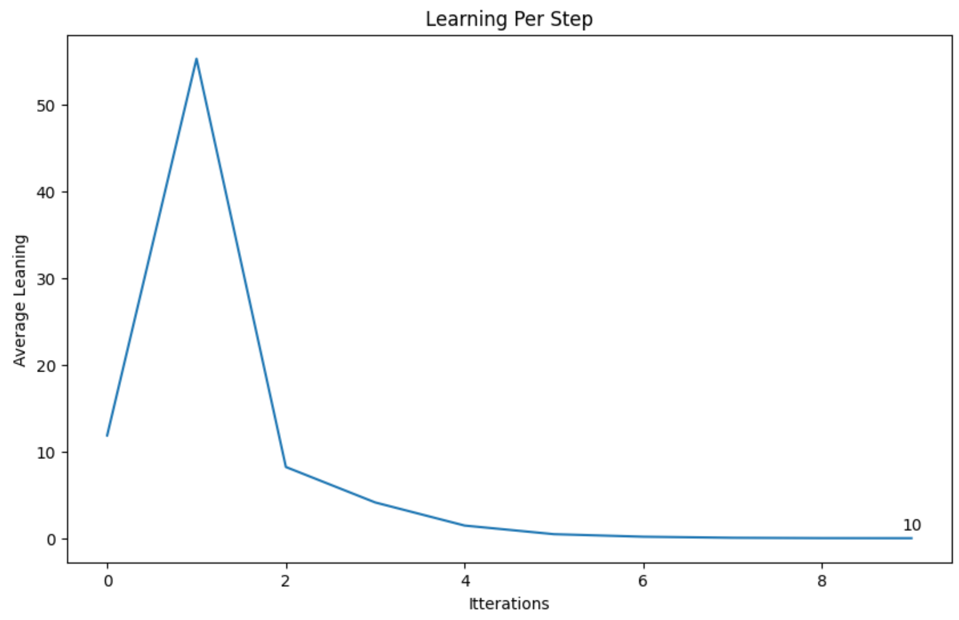


Figure 3.7: Learning rate of large grid using value iteration

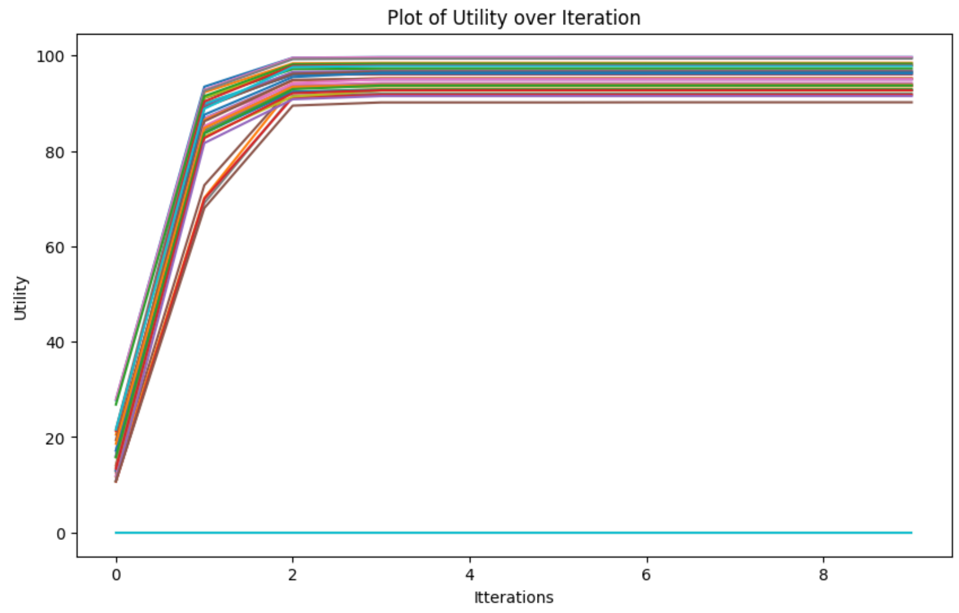


Figure 3.6: Change in utility of large grid using value iteration

From the above figures, it is shown that the utility also converges for a large maze.

Part 2.2 - States, Complexity and Convergence

Set Up and Helper Functions

We set up 3 helper functions to map how different methods converge for different complexity.

1. A function to make square grids with a given width of the grid.
2. A function to solve via value iteration.
3. A function to solve via policy iteration.

The grid is generated with 1/8 as walls, 1/8 as “+1” and 1/8 as “-1” of all the cells.

```
def makeGrid(width):
    def gencoordinates(w, h):
        w=w-1
        h=h-1
        seen = set()
        x, y = random.randint(0, h), random.randint(0, w)
        while True:
            seen.add((x, y))
            yield (x, y)
            x, y = random.randint(0, h), random.randint(0, w)
            while (x, y) in seen:
                x, y = random.randint(0, h), random.randint(0, w)

    WALLS = []
    PLUSONE = []
    MINUSONE = []

    numCells = width*width
    g = gencoordinates(width, width)

    for i in range(numCells//8):
        WALLS.append(next(g))
    for i in range(numCells//8):
        PLUSONE.append(next(g))
    for i in range(numCells//8):
        MINUSONE.append(next(g))

    (grid,reward) = makeGridReward(width, width, PLUSONE, MINUSONE, WALLS)
    return (grid, reward, WALLS)
```

Code 4.1: Grid maker function

```
def solveViaValueIterate(grid, reward, walls):
    height = len(grid)
    width = len(grid[0])
    U = np.zeros((height, width))
    oldU = np.zeros((height, width))
    deltaChange = 1
    deltaHistory = []
    allU=[]

    while deltaChange >pow(10,-5): # converges when absolute sum of difference is > 10^-5
        oldU = U

        U = iterateValue(U, reward, grid, walls)
        deltaChange = np.average(np.absolute(oldU - U))
        deltaHistory.append(float(deltaChange))

        allU.append(U)
        tU = U.copy()
        tU[tU == 0] = np.nan

    return deltaHistory
```
Code 4.2: Solver via value iteration

```
def solveViaPolicyIterate(grid, reward, walls):
    height = len(grid)
    width = len(grid[0])
    U = np.zeros((height, width))
    P = improvePolicy(U, reward, grid, walls)
    oldU = np.zeros((height, width))
    deltaChange = 1
    deltaHistory = []
    stable = False

    while not stable: # converges when absolute sum of difference is > 10^-5
        oldU = U.copy()
        oldP = P.copy()

        U = evaluatePolicy(P, U, reward, grid, walls)
        P = improvePolicy(U, reward, grid, walls)

        if(np.array_equal(oldP, P)):
            stable = True

        deltaChange = np.average(np.absolute(oldU - U))
        deltaHistory.append(float(deltaChange))

    return deltaHistory
```
Code 4.3: Solver via policy iteration

The implementation for code 4.2 and 4.3 has been covered in part 1.

Comparing Value Iteration and Policy Iteration with Maze Complexity

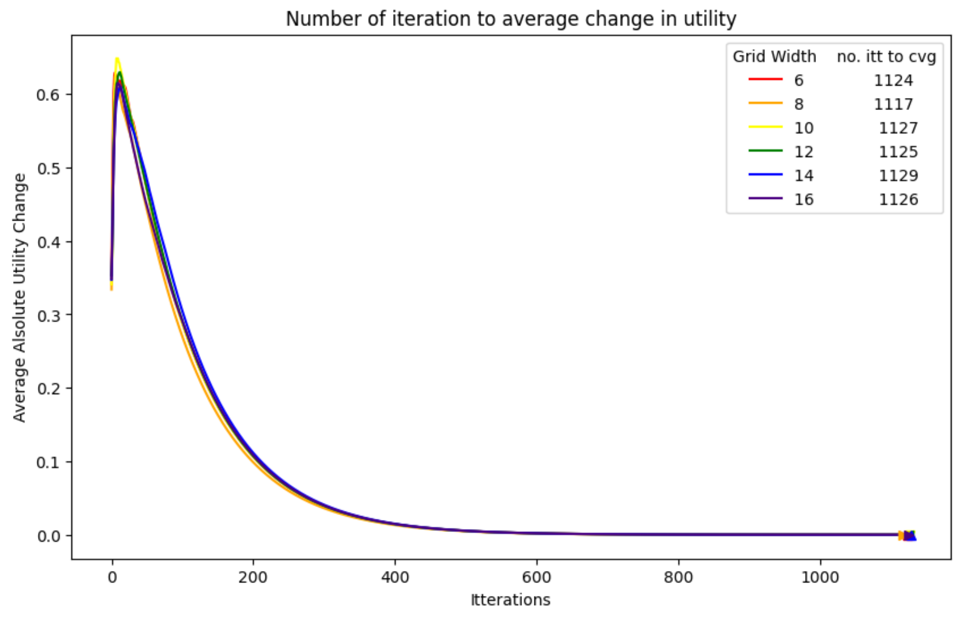


Figure 4.4: Learning rate of different maze complexity via value iteration

From figure 4.4, it is observed that any size of maze from 6x6 to 16x16, the number of iterations to converge stayed consistent. As value iteration was quite slow, I stopped at 16x16.

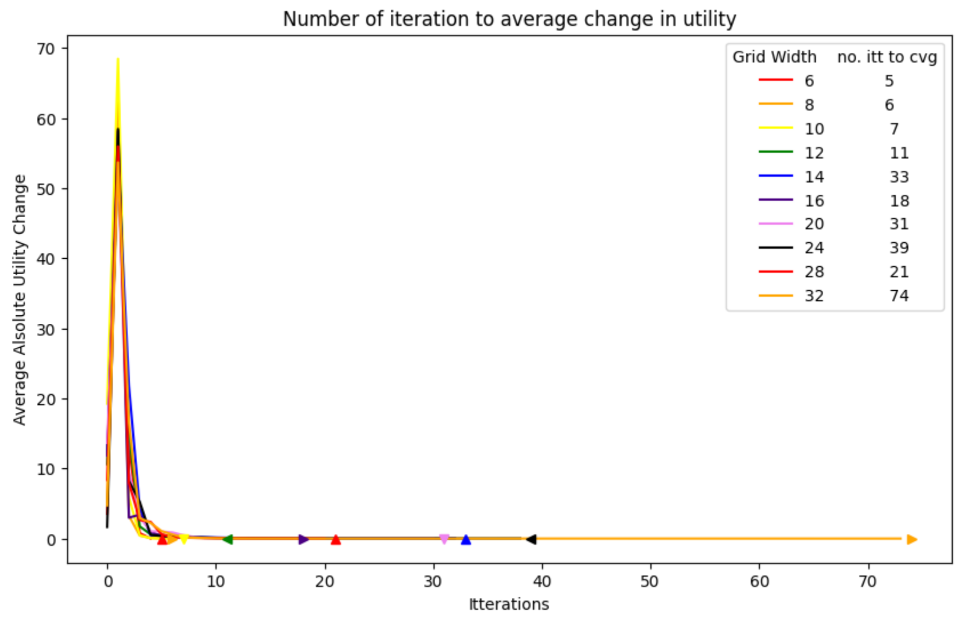


Figure 4.5: Learning rate of different maze complexity via value iteration

From figure 4.5, it is observed that any size of maze from 6x6 to 32x32, the number of iterations seem to fluctuate wildly.

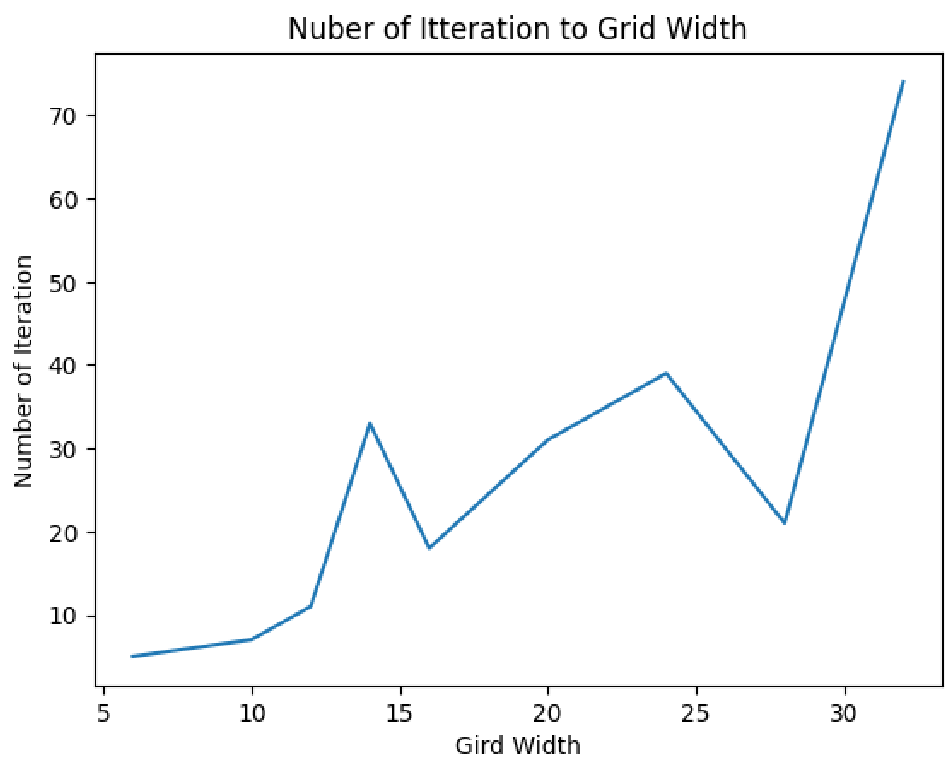


Figure 4.6: Number of iterations compared to grid width using policy iteration

There is still a weak correlation that suggests more complex maze takes more iterations. This is probably due to it taking more iterations for a policy change to propagate to other policy to changes.

From figure 4.5 and figure 4.6, it suggests that all sizes of grid would eventually converge using both methods.

Bibliography

Russell, S. J., & Norvig, P. (2010). *Artificial Intelligence: A modern approach*. Prentice-Hall.

Tokuç, W. by: A. A. (2022, November 9). Value iteration vs. policy iteration in reinforcement learning. Baeldung on Computer Science. Retrieved March 18, 2023, from https://www.baeldung.com/cs/ml-value-iteration-vs-policy-iteration