NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

CZ4046 Intelligent Agents

Assignment 1

Name: Lim Kai Sheng

Matriculation No.: U2020321C

Table of Content

Part 1

1. Value Iteration

1.1 Descriptions of implemented solutions

Value iteration is the finding of the optimal value function, and using it to derive the policy extraction. Once the value function is optimal, we can easily derive the optimal policy. It should also be optimal (i.e. converged).

We do so by maximizing utility through recursion until it converges.

Algorithm (recursion):

1. Initialize all the grid **Utility value V*(s)** to 0
2. Starting from Start state s, do the Bellman update:

   $$V_*(s) = \mathbf{max} \sum_{s'} P(s' \mid s , \pi(s)) [R(s, a, s') + \gamma V_*(s')]$$

3. V*(s) is recursion → keeps updating for every state till reach termination condition
4. If $|V^{*\prime}[s] - V^*[s]| > \delta \rightarrow \delta = |V^{*\prime}[s] - V^*[s]|$
   i. Everytime Bellman update, it measures change in utility estimate for the state:$|V^{*\prime}[s]-V^*[s]|$
   ii. If utility change is larger than current max $\boldsymbol{\delta}$, then update $\boldsymbol{\delta}$ to this new max
5. **Terminal Condition:** until $\delta < \epsilon(\mathbf{1} - \gamma) / \gamma$
   i. $\boldsymbol{\delta}$: change
   ii. $\epsilon(\mathbf{1} - \gamma) / \gamma$: threshold

In short, we start off with utility 0 for all states, and a random value function (Action.UP). Afterwards, we find a new (improved) value function in each iterative process and update the states, until we reach the optimal value function. This process is based on the *Bellman Equation*.

```python
def ValueIteration(maze: Maze):
    threshold = Epsilon * ((1 - gamma) / gamma)  # Convergence threshold
    maxChangeInUtility = 0 # To see improvement (Should get lesser with each iteration)
    iteration = 1 # To count the number of iterations
    logger = LogData(maze) # To log data for graph plotting

    print("Default:")
    maze.printer()
    logger.add(maze)

    while True:
        print("Iteration: {}".format(iteration))
        maxChange = 0

        # Explore the entire maze
        for r in range(NUM_ROW):
            for c in range(NUM_COL):
                currGrid = maze.getGrid(Coordinate(r, c))

                # 1. Skip if currGrid is a wall
                if (currGrid.getGridType() == GridType.WALL):
                    continue

                # 2a. Calculate and update utility on current grid
                # 2b. Get the change in current utility
                currChange = calculateUtility(currGrid, maze)

                # 3. Update the maxChange in utility
                if (currChange > maxChange):
                    maxChangeInUtility = currChange

        iteration += 1
        # Change in utility should get lesser with each iteration
        print("Maximum change in utility: {:.3f}".format(maxChangeInUtility))
        maze.printer() # Print the current maze progress
        logger.add(maze) # Log the data for plotting

        if maxChangeInUtility <= threshold:
            break

    # pprint(logger.data)
    plot(logger.data, file_name)
```

Figure 1.1: “Value Iteration” algorithm

In Figure 1.1, the crucial variables to note are Threshold, Epsilon and Gamma.

1. Epsilon is derived from the multiplication of c*Rmax, where C is variable (changeable), and Rmax is a constant (+1 Reward). Epsilon represents the max error allowable.
2. Discount factor, Gamma, here is 0.99.
3. Together, they are used to calculate the threshold. This threshold will be used for our terminal condition of the algorithm (iteratively).

The thought process and logic flow are elaborated in the comments of Figure 1.1.

```python
# Calculate the utility of the given Grid.
# return The difference prevUtility and newUtility

def calculateUtility(currGrid: Grid, maze: Maze) -> float:
    subUtilities = ['Temp', 'Temp', 'Temp', 'Temp']

    # 1. Find all possible utilities based on intended direction (i.e. UP, DOWN, LEFT, RIGHT)
    for dir in range(4):
        # Sum up the 3 possible utility of each intended direction (i.e. UP, LEFT, RIGHT)
        # 0.8 chance forward, 0.1 chance left, and 0.1 chance right
        neighbours = maze.getNeighboursOfGridwDirection(currGrid, dir)
        up = PROBABILITY_UP * neighbours[0].getUtility()
        left = PROBABILITY_LEFT * neighbours[1].getUtility()
        right = PROBABILITY_RIGHT * neighbours[2].getUtility()

        subUtilities[dir] = up + left + right

    # 2. Find the maximum possible utility
    maxUtilityIndex = 0
    for u in range(len(subUtilities)): # 4 because of the directions above
        if (subUtilities[u] > subUtilities[maxUtilityIndex]):
            maxUtilityIndex = u

    # 3. Get the current grid reward value
    gridType = currGrid.getGridType()
    currReward = getReward(gridType)
    currUtility = currGrid.getUtility() # for pointer 5

    # 4. Update the utility & policy of current grid to the Max Utility
    newUtility = currReward + DISCOUNT_FACTOR * subUtilities[maxUtilityIndex]
    currGrid.setUtility(newUtility)
    currGrid.setPolicy(maxUtilityIndex)

    # 5. Return the difference of currUtility & newUtility
    return (abs(currUtility - newUtility))
```

Figure 1.2: “ Calculate Utility” algorithm

ValueIteration function iteratively calls on calculateUtility function to update every state’s utility. This function also returns the difference between the existing and new utility. The algorithm does so by finding all four directions’ utility. It then updates the existing state’s utility and policy (the index of the array represents the direction).

The thought process and logic flow are elaborated in the comments of Figure 1.2.

1.2 Plot of optimal policy

+1	Wall	+1			+1
	-1		+1	Wall	-1
		-1		+1	
		Start	-1		+1
	Wall	Wall	Wall	-1	

↑	Wall	←	←	←	↑
↑	←	←	←	Wall	↑
↑	←	←	↑	←	←
↑	←	←	↑	↑	↑
↑	Wall	Wall	Wall	↑	↑
↑	←	←	←	↑	↑

Figure 1.3: Plot of optimal policy for value interaction with C = 0.1

Parameters:

- Discount Factor : 0.99
- Utility Upper Bound (Rmax/(1-γ)) : 100.0
- Max Reward (Rmax) : +1.0
- C (variable): 0.1
- Epsilon (C * Rmax) : 0.1
- Convergence threshold ($\epsilon(1-\gamma)/\gamma$) : 0.1010101

The values for "c" are changeable. As Rmax is fixed, by tuning the value of 'C', we are essentially altering the epsilon and affecting the terminal condition, convergence threshold.

A lower value of C means a lower threshold value. This means there will be a higher number of iterations to reach its terminal condition; as the threshold is lower, changes between utilities are required to be minimal. This results in better-estimated utilities and optimal policy.

When the constant "c" is set to 0.1, the Epsilon value will become 0.1, and the convergence threshold is calculated to be 0.00101. After 688 iterations, the following optimal policy is achieved with the following utilities for all states.

1.3 Utilities of all states

↑	Wall	←	←	←	↑
↑	←	←	←	Wall	↑
↑	←	←	↑	←	←
↑	←	←	↑	↑	↑
↑	Wall	Wall	Wall	↑	↑
↑	←	←	←	↑	↑

99.901	Wall	94.950	93.78	92.561	93.236
98.295	95.786	94.449	94.303	Wall	90.826
96.851	95.49	93.200	93.083	93.010	91.703
95.458	94.357	93.138	91.023	91.723	91.798
94.218	Wall	Wall	Wall	89.458	90.477
92.844	91.636	90.443	89.265	88.48	89.209

Figure 1.4: Value Interaction's Utility with C = 0.1

Parameters:

- Discount Factor : 0.99
- Utility Upper Bound (Rmax/(1-γ)) : 100.0
- Max Reward (Rmax) : +1.0
- C (variable): 0.1
- Epsilon (C * Rmax) : 0.1
- Convergence threshold ($\epsilon(1-\gamma)/\gamma$) : 0.0010101

It is worth noting that we can easily derive the optimal policy from the optimal value function. This process is based on the optimality Bellman operator. The grid's policy will "point" to the route with the highest utility among the 4 directions that it can travel (Top, Down, Left, Right). In short, we first determine the optimal value function to derive the optimal policy.

1.4 Plot of utility estimates as a function of the number of iterations

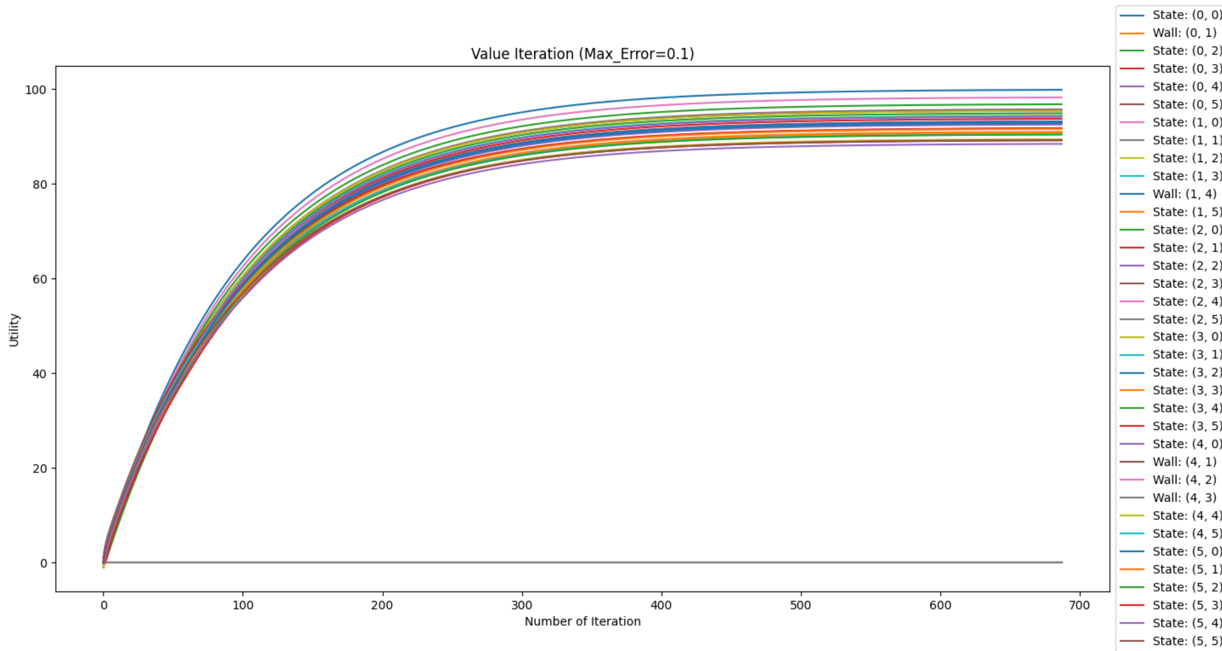

Figure 1.5: Plot of utility estimates against number of iterations with C = 0.1

In this value iteration with C = 0.1, the algorithm requires 687 iterations to reach convergence.

The algorithm will iteratively update the utility value for each state (except wall) using "Bellman update". Theoretically, when the Bellman update is applied infinitely to update the state's utility. The value function will converge and reach stability. The final derived utility function is the unique solution, and the resulting policy is optimal. Since the discount factor is used in "Bellman update", the algorithm is guaranteed to always converge to a unique solution of the Bellman equations whenever the discount factor is less than 1. Therefore, in the above figure, all the estimated utility values gradually increase as the number of iterations increases until it reaches the equilibrium where it starts to level off.

From observation, the 32 state's plotted lines remain relatively the same at about 200 iterations. This means further iterations will not reflect a change in optimal policy. At about 500 iterations, the state's plotted lines reach a stagnant growth rate. In other words, the utility estimates are very close, and an excellent representation of the true utility (where true utility is derived with infinite iteration)

1.5 Part 1 solutions with different C values

↑	Wall	←	←	←	↑
↑	←	←	←	Wall	↑
↑	←	←	↑	←	←
↑	←	←	↑	↑	↑
↑	Wall	Wall	Wall	↑	↑
↑	←	←	←	↑	↑

90.090	Wall	85.480	84.408	83.282	84.050
88.596	86.197	84.98	84.950	Wall	81.732
87.262	86.008	83.823	83.832	83.865	82.665
85.977	84.971	83.847	81.872	82.679	82.849
84.831	Wall	Wall	Wall	80.515	81.619
83.563	82.448	81.347	80.26	79.629	80.44

Figure 1.6: Utilities and Policies for value interaction with C = 10

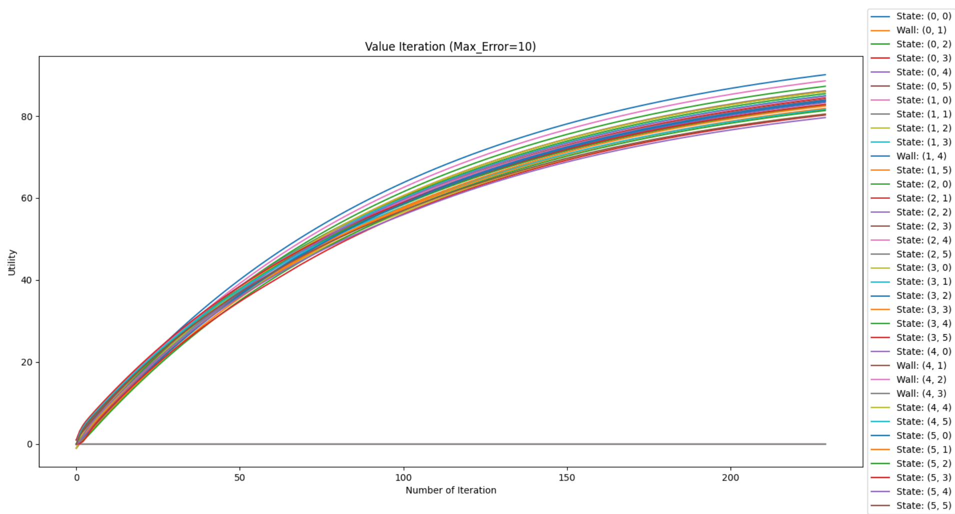

Figure 1.7: Plot of utility estimates against number of iterations with C = 10

In this value iteration with C = 10, the algorithm requires 229 iterations to reach convergence.

This optimal policy (C = 10) is the same as the earlier optimal policy (C = 0.1). This attests to the observation that at about 200 iterations, the state's plotted lines remain relatively the same. The difference between both figures' utility isn't drastic as well. This is reflected in the slowing growth curve.

↑	Wall	←	←	→	↑
↑	←	↑	↑	Wall	↑
↑	←	←	↑	↑	←
↑	←	←	↑	↑	→
↑	Wall	Wall	Wall	↑	↑
↑	←	←	←	→	↑

50.016	Wall	47.613	46.923	46.279	47.484
48.98	47.031	46.98	47.468	Wall	45.533
48.097	47.278	45.592	46.704	47.269	46.522
47.251	46.632	45.903	45.092	46.479	47.076
46.493	Wall	Wall	Wall	44.727	46.204
45.655	44.919	44.194	43.478	44.352	45.396

Figure 1.8: Utilities and Policies for value interaction with C = 50

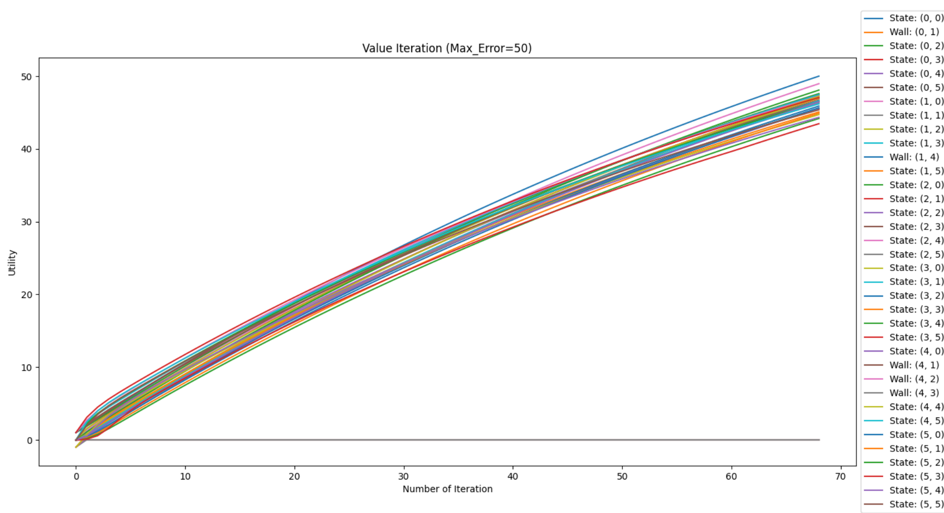

Figure 1.9: Plot of utility estimates against number of iterations with C = 50

In this value iteration with C = 50, the algorithm requires 68 iterations to reach convergence. The differences in optimal policy between (C = 50) and (C = 0.1) are annotated in Figure 1.8.

Suppose one were to trace the route/action taken by the agent. In that case, it makes complete sense to avoid “-1”, head for “+1”, and reduce unnecessary movements. However, this also shows that the agent is shortsighted; it gets short-changed here. This is evident by comparison. This shows that the optimal value function may change by increasing the number of iterations, so does the policy.

2. Policy Iteration

2.1 Descriptions of implemented solutions

In policy iteration, we start with a random policy (Action.UP), then find the value function of that policy using policy evaluation. It does so by iterating "K" numbers of time or until the value converges. Afterwards, we then find a new (improved) policy based on the previous value function, and so on.

Each policy is guaranteed to be a strict improvement over the previous one in this process. And given a policy, its value function can be obtained using the Bellman operator. It is also worth noting that if we were to set K = 1, the policy evaluation will only run once, and will no longer be calling Bellman update iteratively. This converts the algorithm to value iteration instead.

Algorithm (recursion):

1. Start with any arbitrary policy $\boldsymbol{\pi_0}$ (supposed to produce an action given a state, for every possible state)
2. Repeat the following 2 steps iteratively till there's NO CHANGE in the policy $\boldsymbol{\pi_t}$:
    a. Policy Evaluation:
        i. Given a policy $\boldsymbol{\pi_t}$, compute the Utility of starting in state s, given you're following policy $\boldsymbol{\pi_t}$ thereafter : $U_t = U^{\pi t}(s)$ (Bellman Expectation Equation)
        ii. Where $\boldsymbol{U_t(s) = R(s) + \gamma \sum_{s'} P(s' \mid s, \pi_t(s))\, U_t(s')}$
        iii. Iterate simplified bellman update until values converges
    b. Policy Improvement:
        i. Given U_t , find new Maximum Estimated Utility (MEU) policy $\boldsymbol{\pi_{t+1}}$, using the 4 possible directions
        ii. Where $\pi_{t+1}(s) = \text{argmax}_a \sum_{s'} P(s' \mid s, a)\,(R(s, a, s') + \gamma U(s'))$
        iii. Compute the change in current and new utility. Update maxChange δ.
        iv. Terminal Condition: until $\delta < \boldsymbol{\epsilon(1 - \gamma) / \gamma}$ (else, do policy evaluation)
            1. $\boldsymbol{\delta}$: change
            2. $\boldsymbol{\epsilon(1 - \gamma) / \gamma}$: threshold

```python
def PolicyIteration(maze : Maze):
    flag = True
    iteration = 1
    logger = LogData(maze);

    print("Default:")
    maze.printer()
    logger.add(maze)

    # Repeat 1 and 2 until policy is stable
    while flag:
        print("Iteration {}:\n".format(iteration))

        # 1. Policy Evaluation
        policyEvaluation(maze, k)

        # 2. Policy Improvement
        policyStable = True
        # for each s ε S
        for r in range (NUM_ROW):
            for c in range(NUM_COL):
                currGrid = maze.getGrid(Coordinate(r, c))

                # Skip if currGrid is a wall
                if (currGrid.getGridType() == GridType.WALL):
                    continue

                # Run policy improvement algorithm
                changed = policyImprovement(currGrid, maze)

                # current policy != new improved policy
                if (changed):
                    policyStable = False

                # Terminate once policy is stable
                if policyStable:
                    flag = False
                    break

        iteration += 1 # count the number of iterations
        maze.printer() # Print out the progress
        logger.add(maze) # Log data for plotting

    # pprint(logger.data)
    plot(logger.data, file_name)
```

Figure 2.1: "Policy Iteration" algorithm

In Figure 2.1, the critical variable to take note of is K.

1. "K" is the number of times a simplified Bellman update is executed to produce the next utility estimate. A high K value guarantees optimality.
2. If we set K = 1, the Bellman update will only run once instead of iteratively. There will be no value convergence. And thus, the algorithm will implicitly run as value iteration instead.

The thought process and logic flow are elaborated in the comments of Figure 2.1.

```python
def policyEvaluation(maze: Maze, k: int):
    for i in range(k):
        for r in range(NUM_ROW):
            for c in range(NUM_COL):
                # 1. Get current reward & policy
                currGrid = maze.getGrid(Coordinate(r, c))

                # Skip if currGrid is a wall
                if (currGrid.getGridType() == GridType.WALL):
                    continue

                # 2. Sum up the 3 neighbours (i.e. UP, LEFT, RIGHT) based on the current policy
                neighbours = maze.getNeighboursOfGrid(currGrid)
                up = PROBABILITY_UP * neighbours[0].getUtility()
                left = PROBABILITY_LEFT * neighbours[1].getUtility()
                right = PROBABILITY_RIGHT * neighbours[2].getUtility()

                # 3. Update Utility
                gridType = currGrid.getGridType()
                reward = getReward(gridType)
                currGrid.setUtility(
                    reward + DISCOUNT_FACTOR * (up + left + right))
```

Figure 2.2: “Policy Evaluation” algorithm

```python
def policyImprovement(currGrid: Grid, maze: Maze) -> bool:
    # 1. Find the maximum possible sub-utility
    maxSubUtility = ['Temp', 'Temp', 'Temp', 'Temp']
    for dir in range(4):
        neighbours = maze.getNeighboursOfGridwDirection(currGrid, dir)
        up = PROBABILITY_UP * neighbours[0].getUtility()  # front grid utility
        left = PROBABILITY_LEFT * neighbours[1].getUtility()  # left grid utility
        right = PROBABILITY_RIGHT * neighbours[2].getUtility()  # right grid utlity

        maxSubUtility[dir] = up + left + right

    maxSU = 0
    # 2. Get the highest utility "direction"
    for dir in range(1, len(maxSubUtility)):
        if (maxSubUtility[dir] > maxSubUtility[maxSU]):
            maxSU = dir

    # 3. Current sub-utility (based on current policy)
    neighbours = maze.getNeighboursOfGrid(currGrid)
    up = PROBABILITY_UP * neighbours[0].getUtility() # Forward
    left = PROBABILITY_LEFT * neighbours[1].getUtility() # Noise
    right = PROBABILITY_RIGHT * neighbours[2].getUtility() # Noise

    currSubUtility = up + left + right

    # 4. Update policy
        # It will be true if there's a need to change the direction as there's a better utility.
        # After each iteration, this statement should be run lesser and lesser
    if (maxSubUtility[maxSU] > currSubUtility):
        currGrid.setPolicy(maxSU)
        return True
    else:
        return False
```

Figure 2.3: “Policy Improvement” algorithm

PolicyIteration function iteratively calls on both policyEvaluation and policyImprovement function until the policy is stable. Meaning to say, the current policy is the same as the optimal policy for all states. When the policy is stable, it is said that the policy converges.

The thought process and logic flow are elaborated in the comments of Figure 2.2 and Figure 2.3.

2.2 Plot of optimal policy

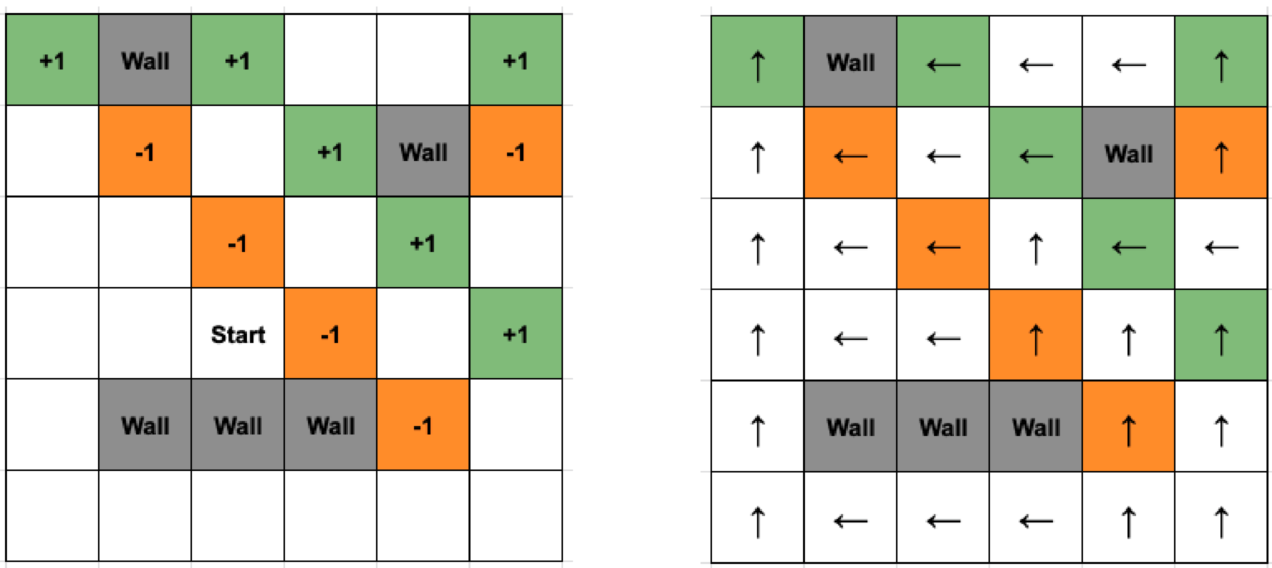

Figure 2.4: Plot of optimal policy for policy interaction with K = 100; 6 Iterations

Parameters:

- Discount Factor : 0.99
- Utility Upper Bound (Rmax/(1-γ)) : 100.0
- Max Reward (Rmax) : +1.0
- K repetition (variable) : 100

The value of “K” is variable. By altering the value of ‘K’, we are essentially changing the number of times policy evaluation (bellman update) needs to run before executing policy improvement. By setting a higher “K” value, it will increase the final estimated utility, which in turn makes the policy more optimized. Here, we set K as 100 (a large value) to get optimality. It performs 6 iterations.

2.3 Utilities of all states

↑	Wall	←	←	←	↑
↑	←	←	←	Wall	↑
↑	←	←	↑	←	←
↑	←	←	↑	↑	↑
↑	Wall	Wall	Wall	↑	↑
↑	←	←	←	↑	↑

99.762	Wall	94.816	93.648	92.429	93.106
98.158	95.650	94.315	94.171	Wall	90.697
96.716	95.356	93.067	92.952	92.880	91.576
95.324	94.225	93.007	90.893	91.595	91.671
94.085	Wall	Wall	Wall	89.331	90.352
92.712	91.506	90.314	89.138	88.354	89.085

Figure 2.5: Utilities for policy interaction with K = 100; 6 Iterations

Here, we are setting K to a huge value so that we can run policy evaluation iteratively to get a higher estimated utility value for value convergence. These generated utilities will be a good reflection of the 36 states and will significantly aid in policy improvement.

2.4 Plot of utility estimates as a function of the number of iterations

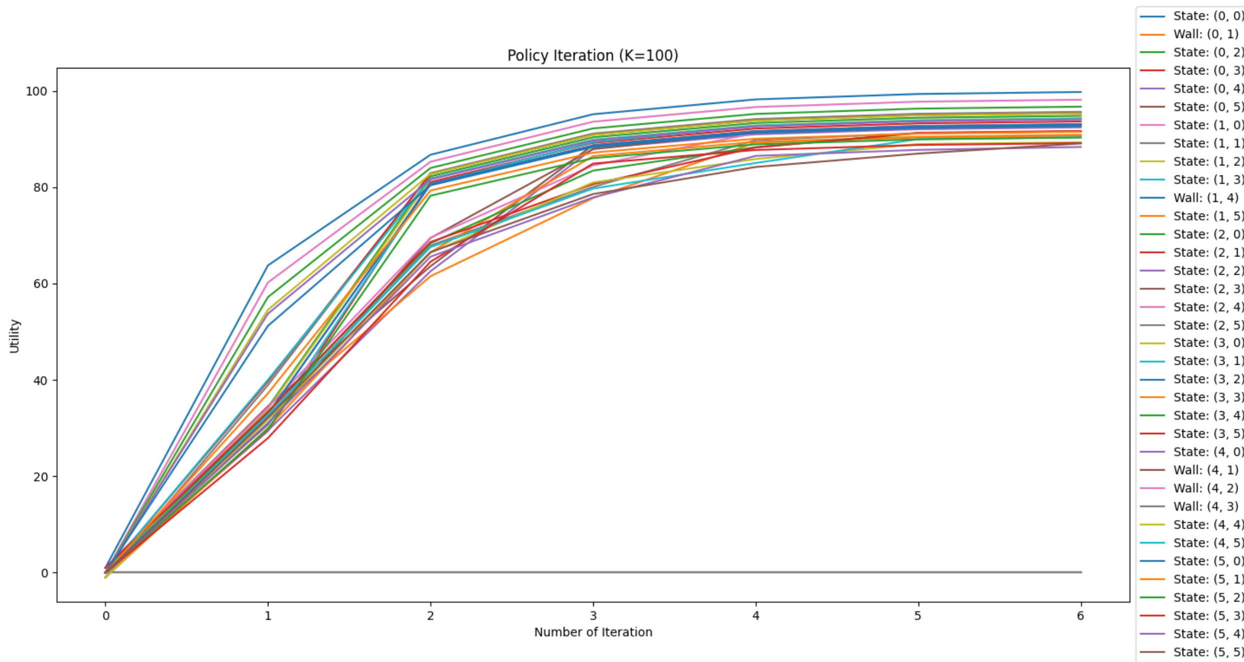

Figure 2.6: Plot of utility estimates against number of iterations with K = 100

As you've already noticed, the number of iterations needed to achieve optimality is much lower. This is because the iteration will terminate once the policy is stable. The core idea behind Policy Iteration is to get a reasonable estimate of the utility by running policy evaluation for K number of times. With more policy evaluation run within each iteration, the state's utility will be a good representation of its true utility. Using these values, the algorithm can update its policy.

After each iteration, the agent is guaranteed to have a policy improvement because it can recognize which action for a particular state is clearly more optimal than the other. Hence, the exact value of the utility for that action holds little significance. And this is what differentiates between value iteration and policy iteration. Value iteration works on the principle of getting the optimal value function before deriving the optimal policy. In comparison, Policy iteration works on doing Policy evaluation and Policy improvement iteratively until the policy is stable.

2.5 Part 1 solutions with different K values

↑	Wall	←	←	←	↑
↑	←	←	←	Wall	↑
↑	←	←	↑	←	←
↑	←	←	↑	↑	↑
↑	Wall	Wall	Wall	↑	↑
↑	←	←	←	↑	↑

95.145	Wall	90.359	89.237	88.063	88.783
93.594	91.138	89.859	89.769	Wall	86.418
92.203	90.894	88.654	88.599	88.577	87.322
90.862	89.808	88.635	86.587	87.339	87.460
89.668	Wall	Wall	Wall	85.123	86.183
88.345	87.182	86.034	84.9	84.189	84.958

Figure 2.7: Utilities and Policies for policy interaction with K = 25; 12 Iterations

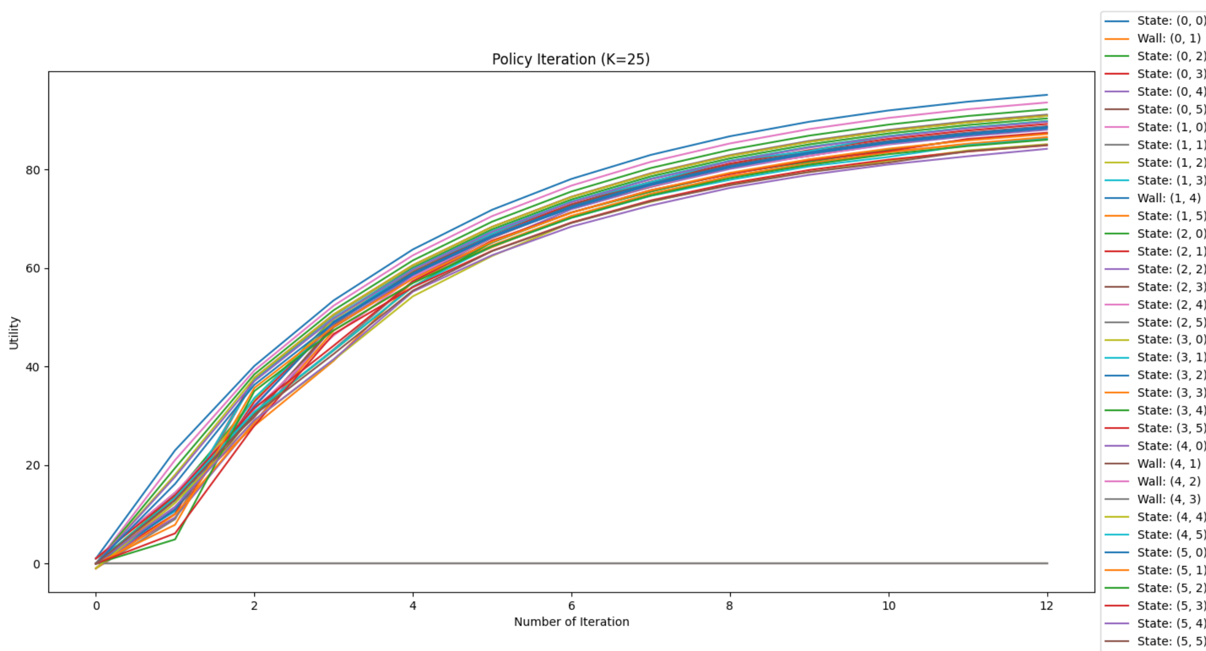

Figure 2.8: Plot of utility estimates against number of iterations with K = 25

In this policy iteration with K = 25, the algorithm requires 12 iterations to reach convergence. The optimal policy (K = 25) is the same as the earlier optimal policy (K = 100). This attests to the earlier statement that we can get an optimal policy even with a lower K value, so long as the utility is a reasonable estimate of its true utility value. The algorithm will be able to update the policy better and differentiate which action is better than the others.

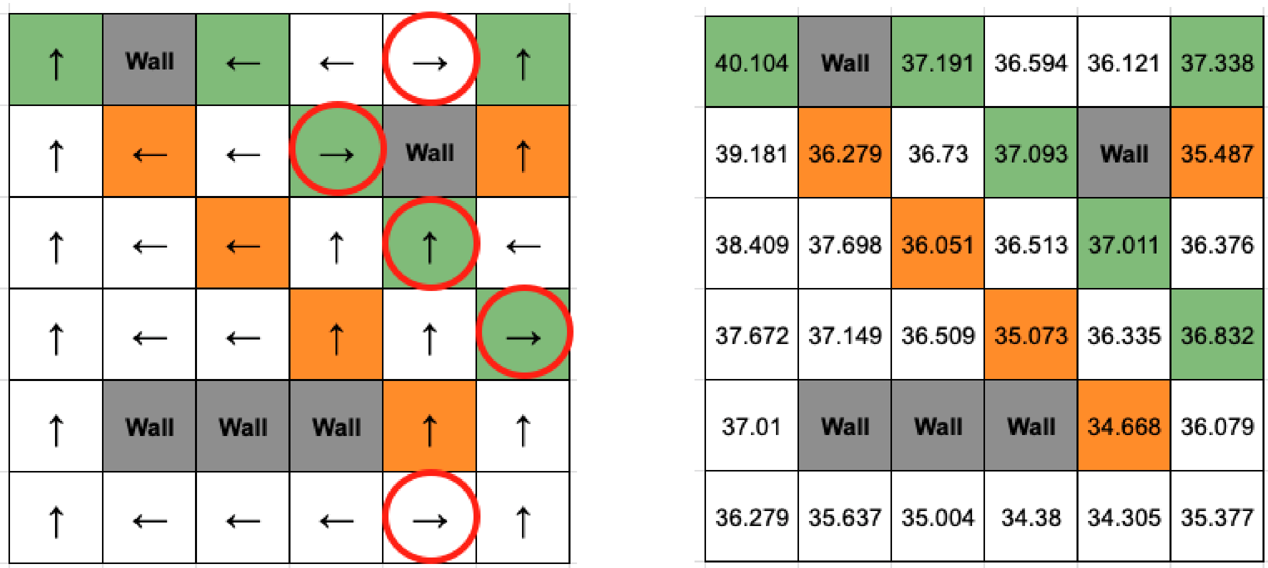

Figure 2.9: Utilities and Policies for policy interaction with K = 5; 10 Iterations

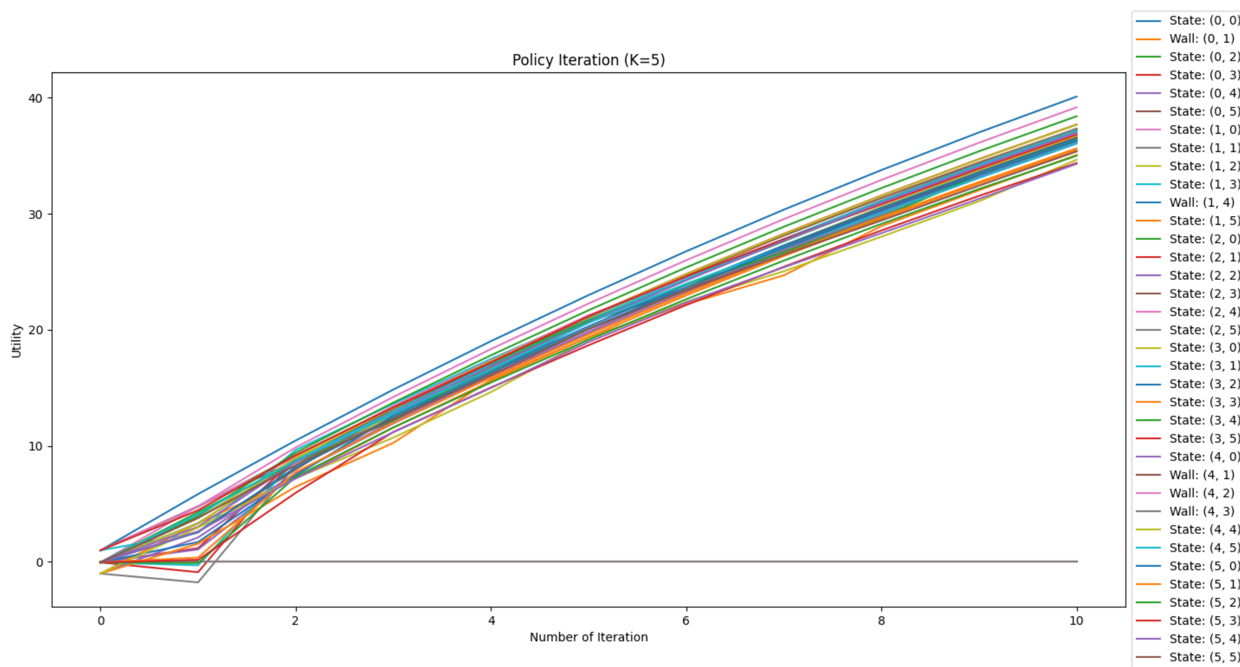

Figure 2.10: Plot of utility estimates against number of iterations with K = 5

In this policy iteration with K = 5, the algorithm requires 10 iterations to reach convergence. The differences in optimal policy between (K = 100) and (K = 5) are annotated in Figure 2.9. As shown, a low K value means lesser iteration of policy evaluation, resulting in a poor representation of the state's true utility value. Thus, leading to a different and not so optimal policy.

3. Source code

3.1 File Structure

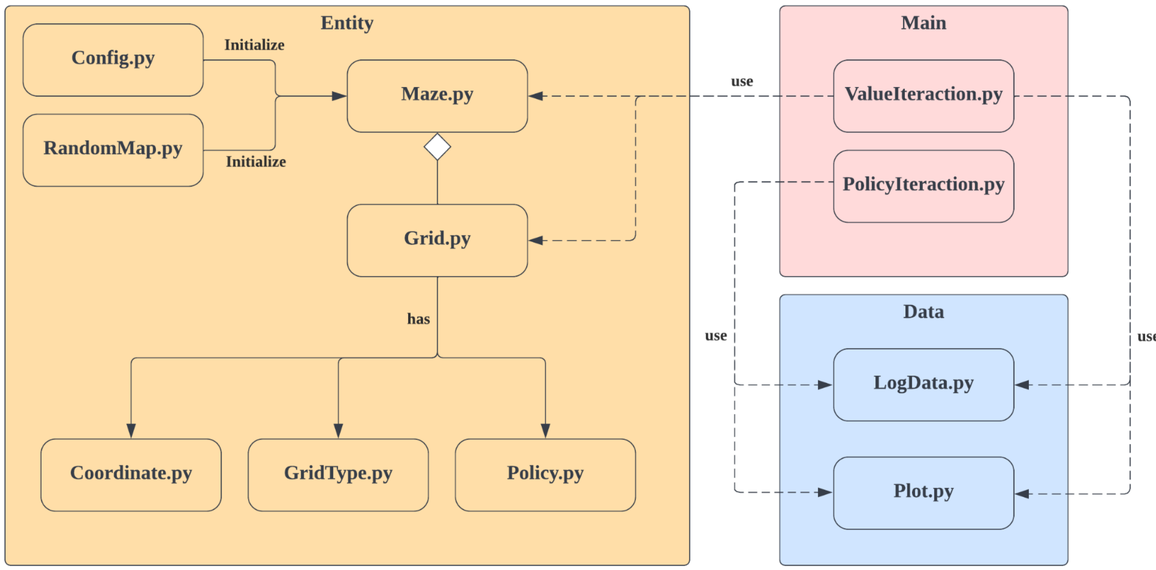

Figure 3.1: Schema of the file structure with arrows indicating files' dependency

We employed object-oriented programming to structure this assignment into smaller components, each with its attributes and functions. This makes the code more understandable, as each object now serves to carry out its purpose. The objects are then integrated with the "main" algorithm for "Value Iteration" and "Policy Iteration".

Main Package:

1. ValueIteration.py → The executable file for the value iteration. (Variable C value)
2. PolicyIteration.py → The executable file for the value iteration. (Variable K value)

Entity Package:

1. Config.py → For initialization of Maze.py (Constants, Default and Random Maze)
2. RandomMap.py → Generate Random Maze for Part 2
3. Maze.py → Aggregation of grids; To get specified and neighboring grids
4. Grid.py → Has Policy, Utility, GridType. These are then broken into subcomponents
5. Policy.py → An enumeration of directions that agent can take
6. GridType.py → An enumeration of grid type and its reward
7. Coordinate.py → To move about the maze, using coordinates increment/decrement

Data Package:

1. LogData.py → Dictionary to store the state's corresponding utility value after each iteration
2. Plot.py → Using pyplot to generate graph using the data logged from the main algorithm

Part 2

	0	1	2	3	4	5	6	7	8	9	10	11
0	Wall	Wall	+1					+1		+1	+1	
1	-1	+1	Wall	-1	+1		+1		+1	+1		Wall
2		-1	Wall		Wall				-1	+1		+1
3	-1		+1			+1	Wall	-1	Wall		Wall	-1
4	+1	-1	-1				Wall	+1	Wall	+1	Wall	-1
5	Wall	-1		+1	-1	+1	+1	Wall	Wall		Wall	-1
6		Wall	-1		-1	-1	Wall	Wall	+1	-1	+1	+1
7	-1	+1	+1	+1	Wall	Wall	-1	Wall			+1	+1
8	Wall	Wall	+1	+1					+1	Wall	-1	Wall
9	Wall	-1	+1	-1	-1			Wall		-1	+1	Wall
10	Wall	-1		+1	Wall	+1	+1	+1	-1	Wall	+1	-1
11			Wall	Wall	Wall	-1	-1	Wall	-1	-1	Wall	

Figure 4.1: Complex Maze generated using randomMap.py

We can create a complex maze environment by using randomMap.py. Here, we increase the maze size to twice its size (12), and each gridType is randomly assigned using a randomizer.

1. Value Iteration

1.1 Plot of optimal policy and utilities

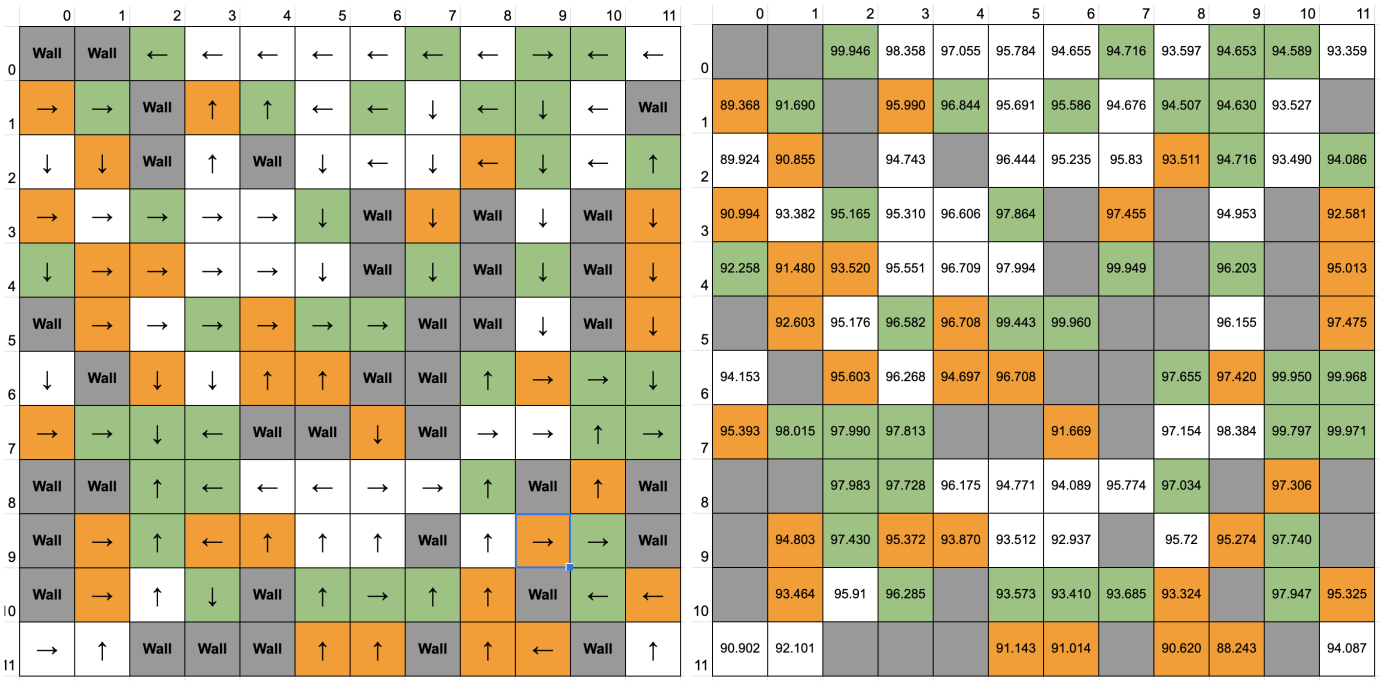

Figure 4.1: Value interaction with C = 0.01; 718 Iterations

1.2 Plot of utility estimates as a function of the number of iterations

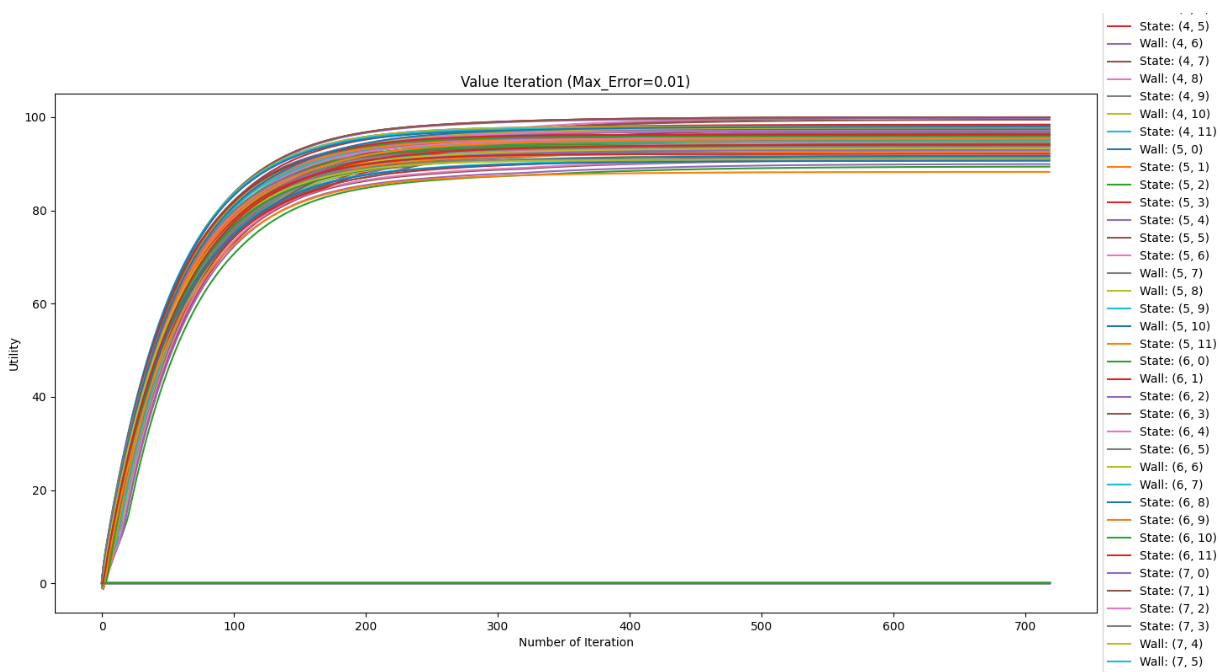

Figure 4.2: Plot of utility estimates against number of iterations with C = 0.01

2. Policy iteration

2.1 Plot of optimal policy and utilities

	0	1	2	3	4	5	6	7	8	9	10	11
0	Wall	Wall	←	←	←	→	→	→	→	→	←	←
1	→	→	Wall	↑	←	→	→	→	→	→	↑	Wall
2	↓	↓	Wall	↑	Wall	↓	↑	↑	↑	↑	↑	↑
3	↓	→	→	↑	↓	↓	Wall	↓	Wall	↓	Wall	↑
4	↓	→	→	↓	→	↓	Wall	↓	Wall	→	Wall	↑
5	Wall	→	→	→	→	→	→	Wall	Wall	↓	Wall	↓
6	↓	Wall	→	↑	→	↑	Wall	Wall	←	←	↓	↓
7	→	→	↓	→	Wall	Wall	↓	Wall	↑	←	↑	↓
8	Wall	Wall	↑	↑	←	←	←	→	→	Wall	↓	Wall
9	Wall	→	↑	↑	↑	↑	←	Wall	↑	→	↓	Wall
10	Wall	↑	↑	↑	Wall	↑	←	←	↑	Wall	←	←
11	→	↑	Wall	Wall	Wall	↑	←	Wall	↑	←	Wall	↑

	0	1	2	3	4	5	6	7	8	9	10	11
0			100.000	98.4	97.012	82.654	83.931	85.213	85.315	86.551	86.573	85.444
1	76.108	78.300		95.947	96.109	81.197	82.557	82.900	84.450	85.013	85.396	
2	76.361	76.108	94.700			77.466	76.677	76.96	77.516	78.891	77.924	78.024
3	86.756	90.014	92.912	92.881	88.084	88.283		74.754		78.368		75.805
4	89.954	90.227	92.445	94.555	95.795	97.916		100.000		79.408		73.613
5		91.906	94.539	96.263	96.620	99.468	100.000			78.368		71.448
6	91.188		92.792	94.744	94.517	96.710			87.281	84.281	85.365	84.145
7	92.390	94.969	94.906	95.100			81.703		85.996	84.809	85.468	85.484
8			94.951	94.961	93.347	91.815	83.998	84.994	86.117		83.155	
9		91.661	94.412	92.287	90.269	89.24	87.211		84.836	83.560	85.878	
10		89.012	91.23	92.034		87.847	83.406	82.776	82.558		87.173	84.703
11	86.567	87.711				85.226	82.970		80.005	77.761		83.597

Figure 4.3: Value interaction with K = 10000; 2 Iterations

2.2 Plot of utility estimates as a function of the number of iterations

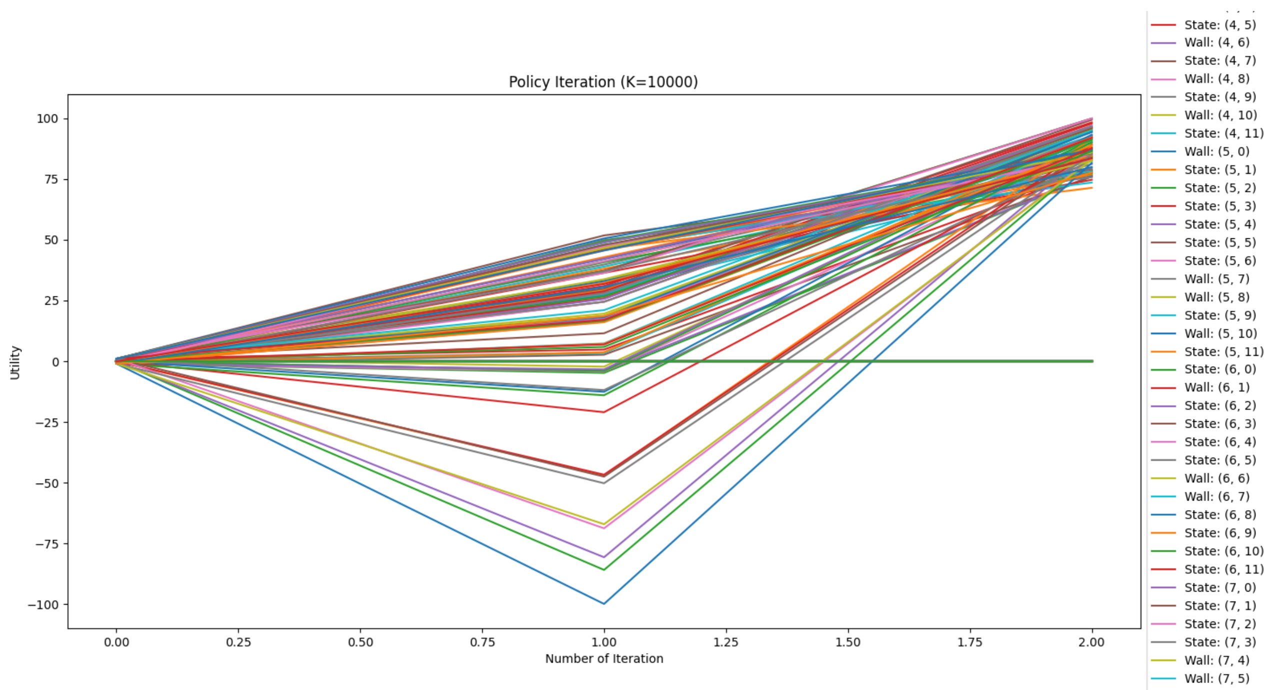

Figure 4.4: Plot of utility estimates against number of iterations with K = 10000

2.3 Analysis between VI and PI policies

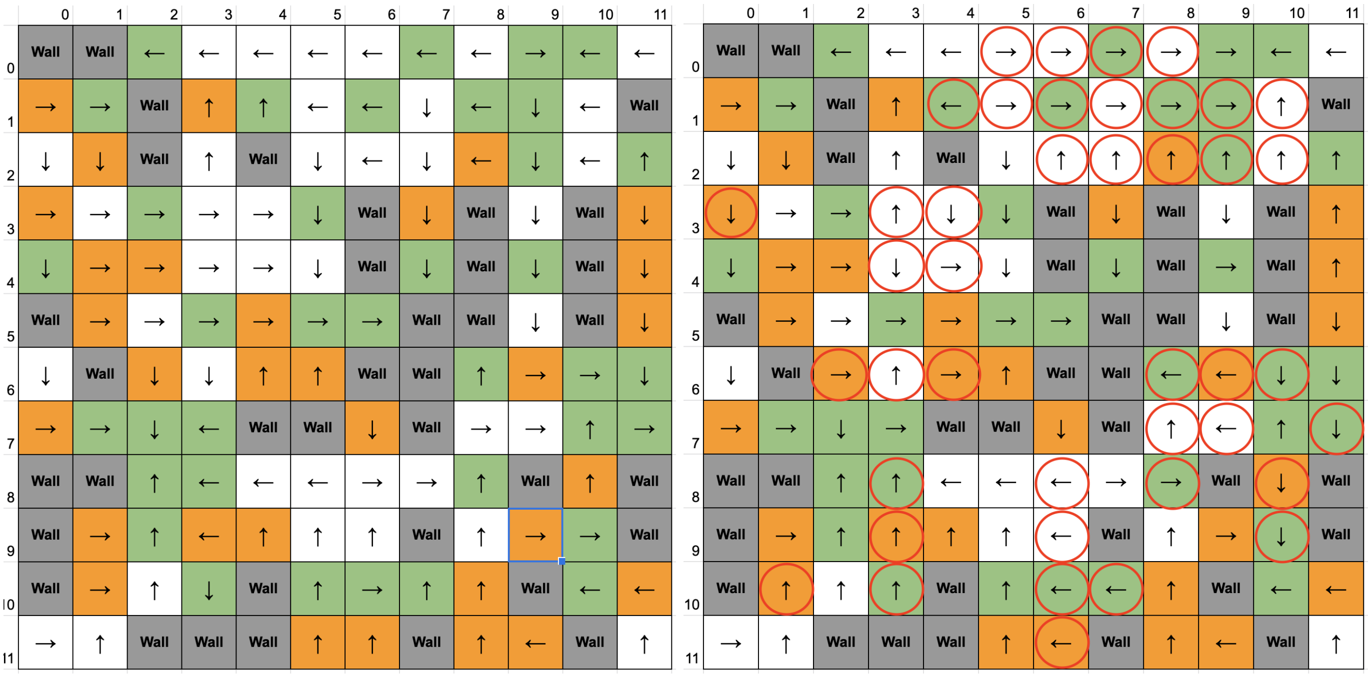

Figure 4.7: Juxtaposition of Policy between Value iteration and Policy Iteration

In this figure, the differences in policy are annotated in red. Both value iteration (VI) and policy iteration (PI) algorithms are guaranteed to converge to an optimal policy, so it is expected to get similar policies from both algorithms (if they have converged). However, if an MDP has several optimal policies, both VI and PI algorithms could converge to any of the optimal policies.