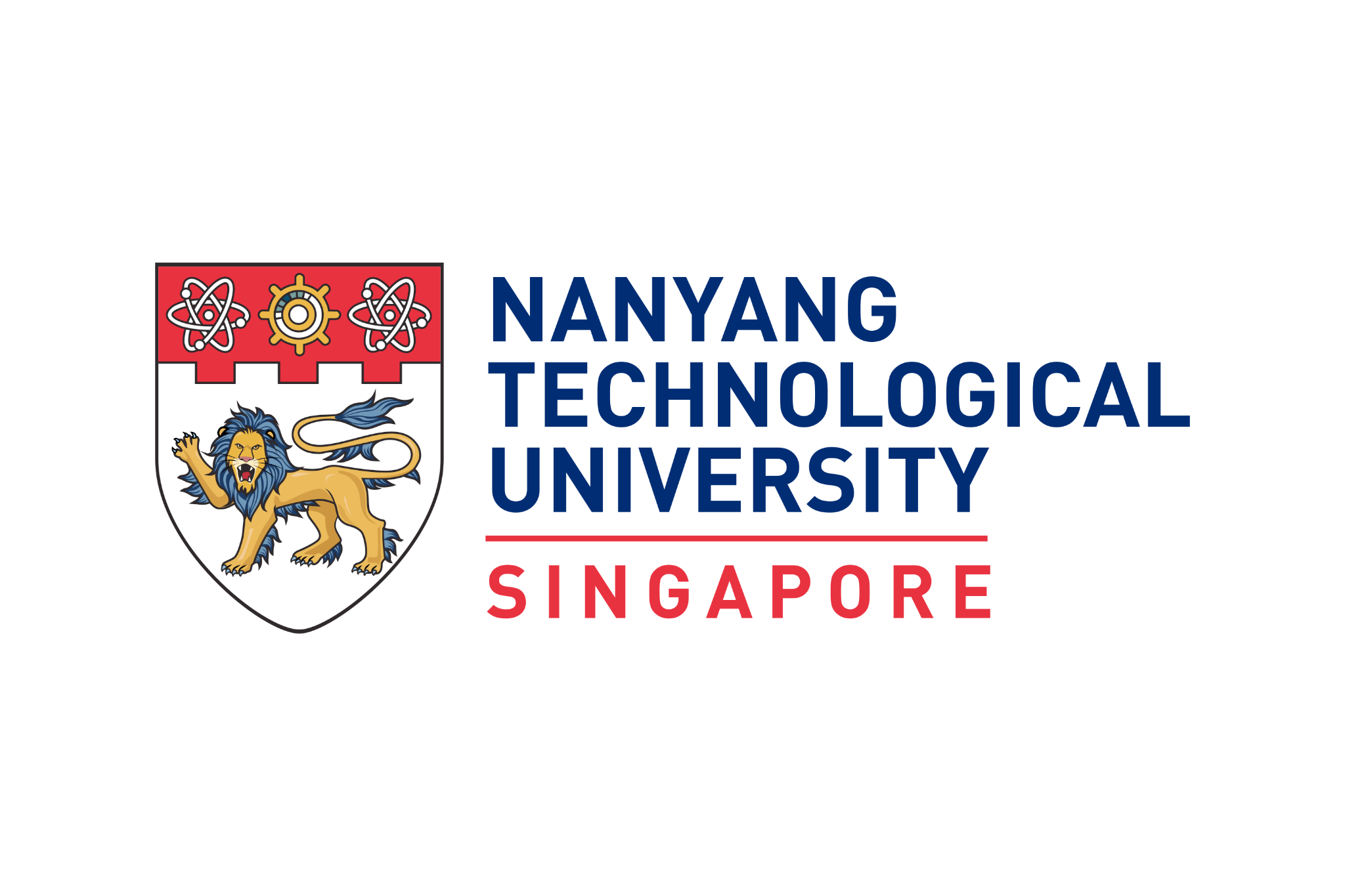
****

**CZ4046 Intelligent Agents**

**Assignment 1**

**Name: Lim Kai Sheng**

**Matriculation No.: U2020321C**

**Table of Content**

[**Part 1**](#_4l0dkqphzbjc) **3**

[1. Value Iteration](#_pusmmfxl0o1f) 3

[1.1 Descriptions of implemented solutions](#_6xx3slb2eye0) 3

[1.2 Plot of optimal policy](#_49fmfbj8exlt) 6

[1.3 Utilities of all states](#_pf8wwf9v4xst) 7

[1.4 Plot of utility estimates as a function of the number of iterations](#_cbmqwd733xjn) 8

[1.5 Part 1 solutions with different C values](#_bn6knht80sry) 9

[2. Policy Iteration](#_jrcmojwurt1q) 11

[2.1 Descriptions of implemented solutions](#_klgwwzxfnsuf) 11

[2.2 Plot of optimal policy](#_d3ppm1kx38li) 14

[2.3 Utilities of all states](#_agg4mhcvz0nb) 15

[2.4 Plot of utility estimates as a function of the number of iterations](#_cdbpfz7d4axb) 16

[2.5 Part 1 solutions with different K values](#_qb267cxzy9i5) 17

[3. Source code](#_6r3kbfu5n9lu) 19

[3.1 File Structure](#_j22r9jdu9el5) 19

[**Part 2**](#_foim1057cpx4) **20**

[1. Value Iteration](#_jy8am5dkm6h5) 21

[1.1 Plot of optimal policy and utilities](#_3hjo9auj73cu) 21

[1.2 Plot of utility estimates as a function of the number of iterations](#_uqeuk2z3pm4q) 21

[2. Policy iteration](#_xg1qwfqlf18o) 22

[2.1 Plot of optimal policy and utilities](#_l7oh07lra68s) 22

[2.2 Plot of utility estimates as a function of the number of iterations](#_n3xl9613trtq) 22

[2.3 Analysis between VI and PI policies](#_n7m79n9s6ksi) 23

# 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) = 𝐦𝐚𝐱 ∑𝒔’ P(𝒔’ | 𝒔 , π(s) ) [𝑹(𝒔, 𝒂, 𝒔’) + 𝜸𝑽∗(𝐬’)]

1. V\*(s) is recursion → keeps updating for every state till reach termination condition
2. If |V\*’[s] – V\*[s]| > δ → δ = |V\*’[s] – V\*[s]|
   * 1. Everytime Bellman update, it measures change in utility estimate for the state:|V\*’[s]–V\*[s]|
     2. If utility change is larger than current max 𝜹, then update 𝜹 to this new max
3. **Terminal Condition:** until δ < 𝝐(𝟏 – 𝜸) / 𝜸
   * 1. 𝜹 : change
     2. 𝝐(𝟏 – 𝜸) / 𝜸 : 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*.



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.

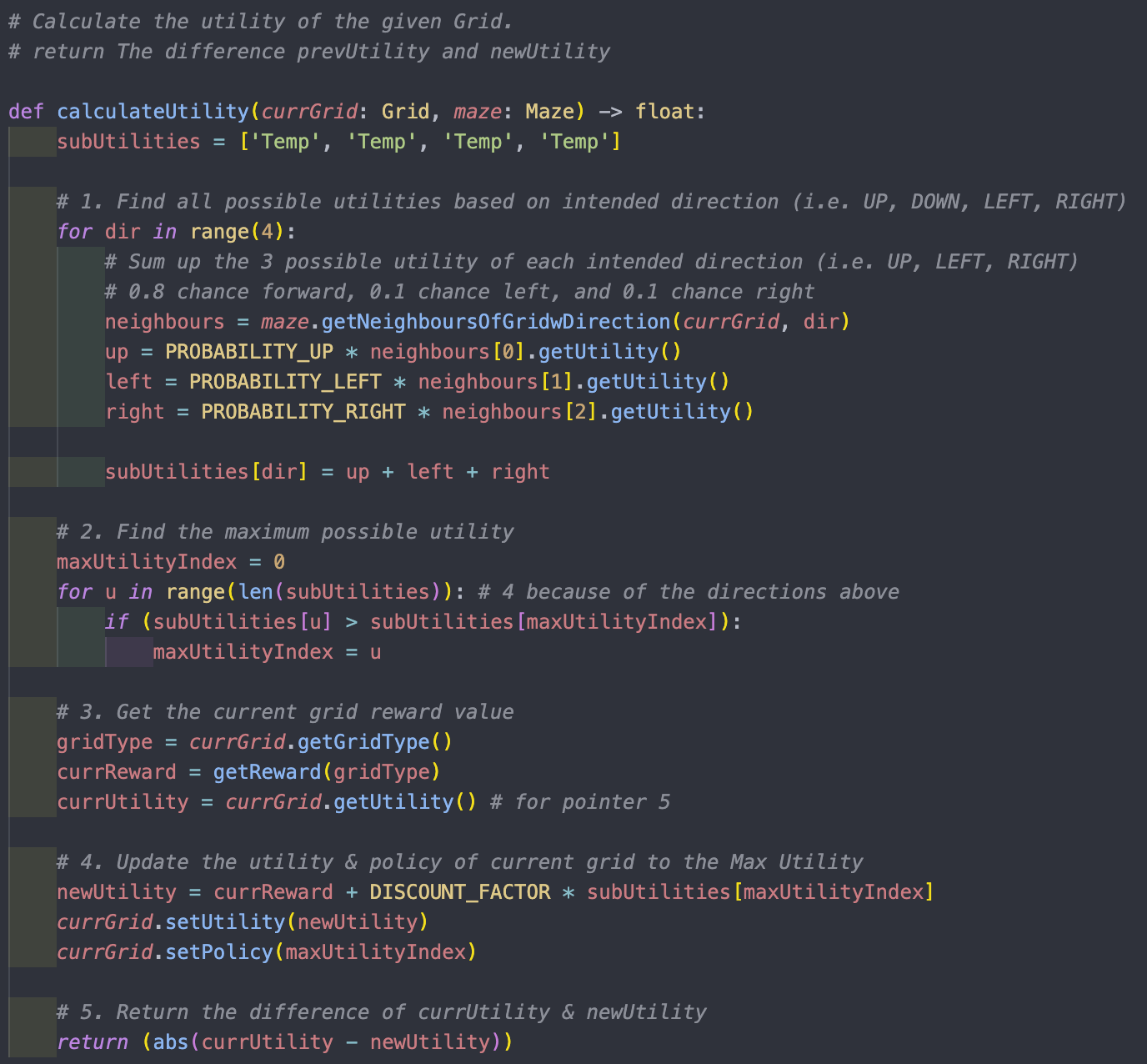


Figure 1.2: “ Calculate Utility” algorithm

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

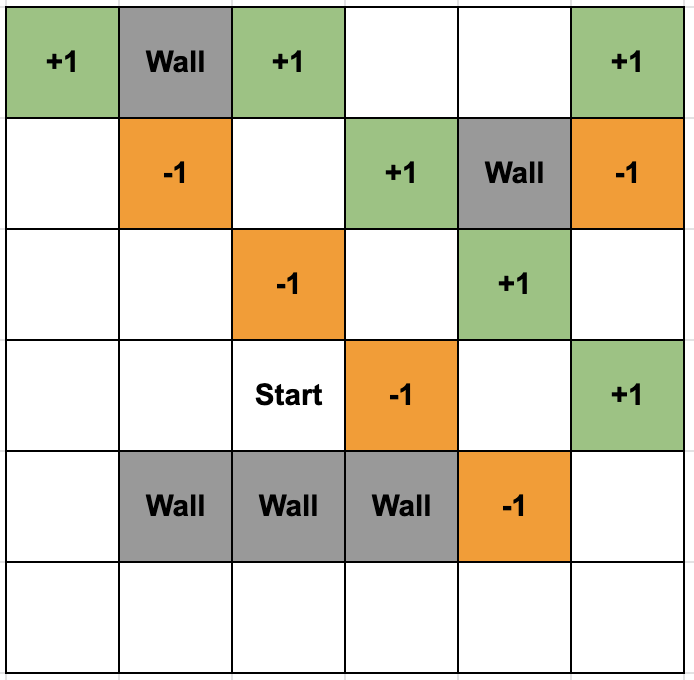
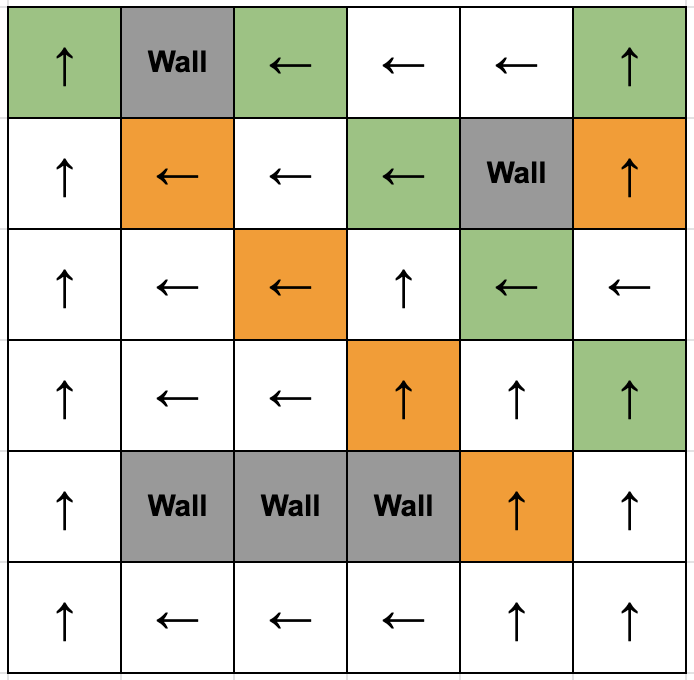
 

Figure 1.3: Plot of optimal policy for value interaction with C = 0.1

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 ( 𝝐(𝟏 – 𝜸) / 𝜸 ) : 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

### 

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 ( 𝝐(𝟏 – 𝜸) / 𝜸 ) : 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

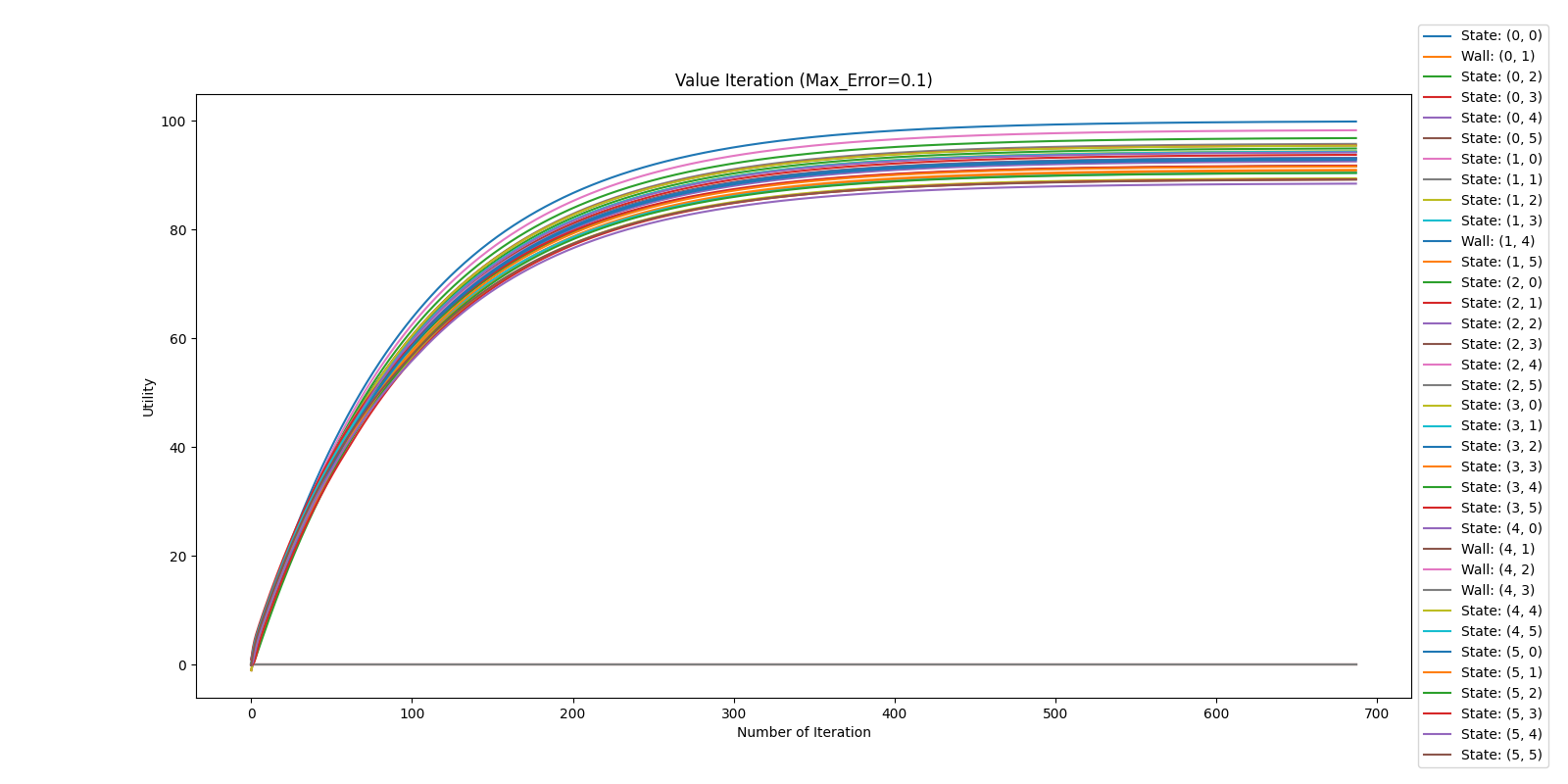


Figure 1.5: Plot of utility estimates against number of iterations with C = 0.1

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

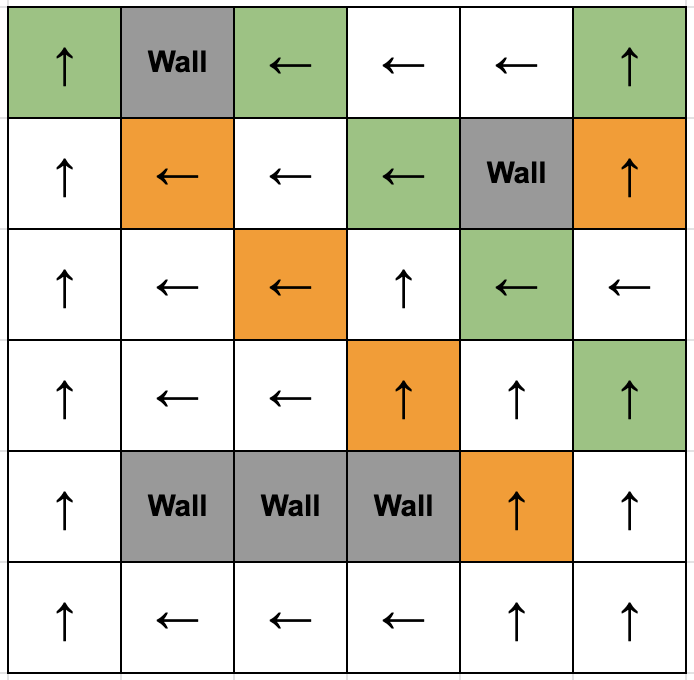
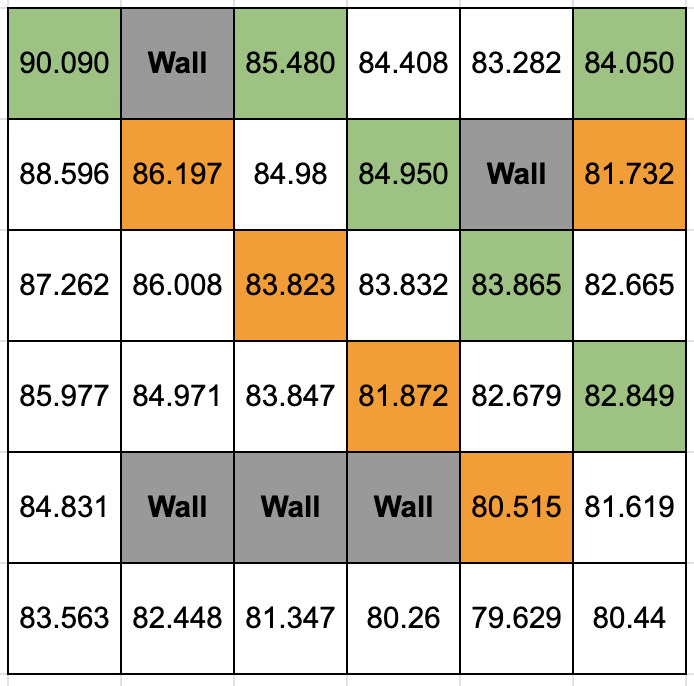
 

Figure 1.6: Utilities and Policies for value interaction with C = 10

### 

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.

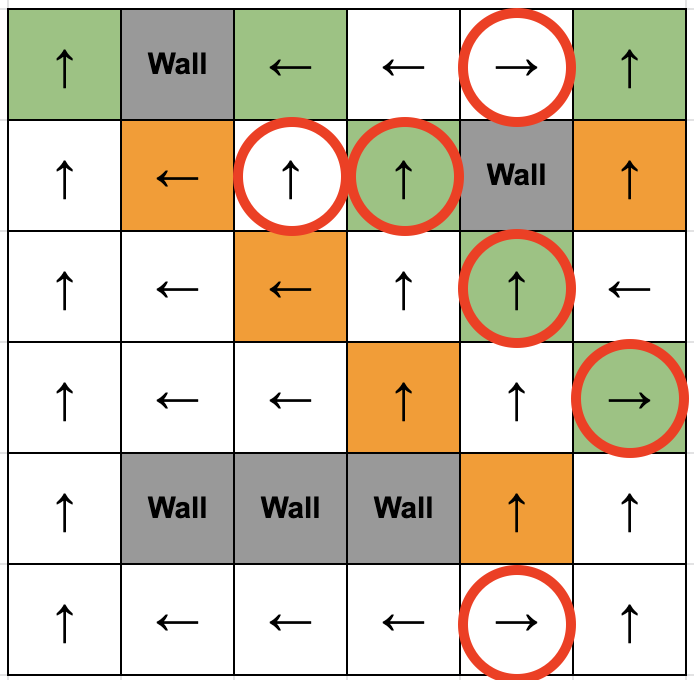
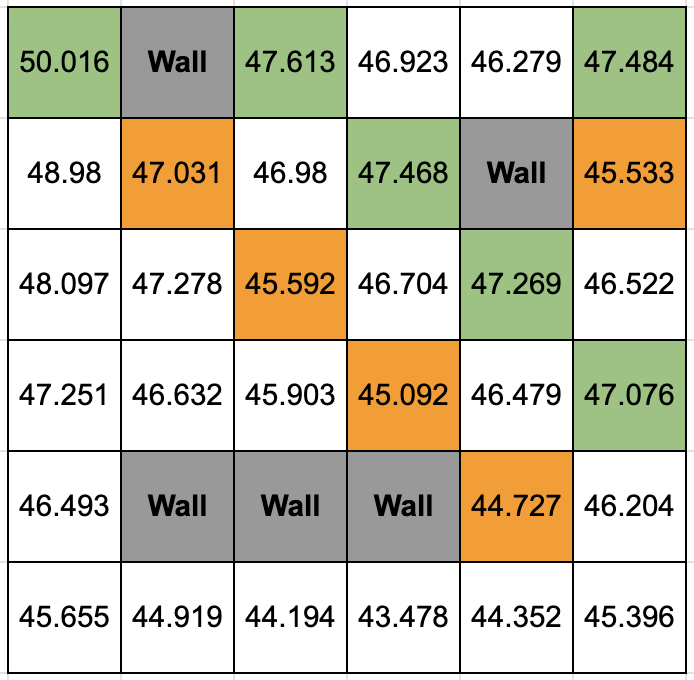
 

Figure 1.8: Utilities and Policies for value interaction with C = 50

### 

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 𝝅𝟎 (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 𝝅𝒕:
   1. Policy Evaluation:
      1. Given a policy 𝝅𝒕 , compute the Utility of starting in state s, given you’re following policy 𝝅𝒕 thereafter : 𝑈t = 𝑈 π t (s) (Bellman Expectation Equation)
      2. Where 𝑼𝒕 (𝒔) = 𝑹(𝒔) + 𝜸 ∑ 𝒔’ ***P***( 𝒔’ | 𝒔, 𝝅𝒕(𝒔) ) 𝑼𝒕(𝒔′)
      3. Iterate simplified bellman update until values converges
   2. Policy Improvement:
      1. Given Ut , find new Maximum Estimated Utility (MEU) policy 𝝅𝒕+𝟏 , using the 4 possible directions
      2. Where π𝒕+𝟏(s) = argmax a ∑ s' P( s' | s, a ) ( R( s, a, s' )+ γU(s'))
      3. Compute the change in current and new utility. Update maxChange δ.
      4. Terminal Condition: until δ < 𝝐(𝟏 – 𝜸) / 𝜸 (else, do policy evaluation)
         1. 𝜹 : change
         2. 𝝐(𝟏 – 𝜸) / 𝜸 : threshold



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.

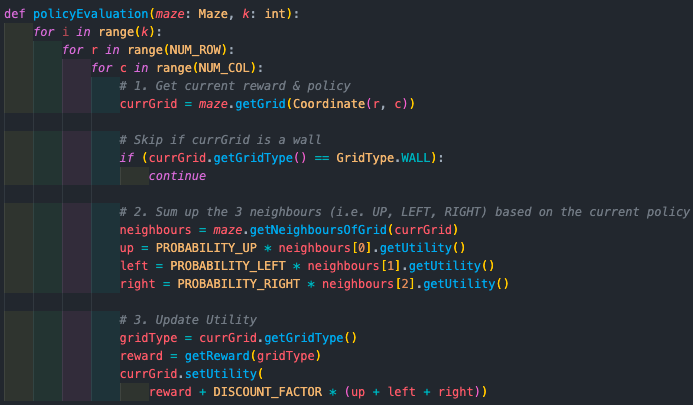


Figure 2.2: “Policy Evaluation” algorithm

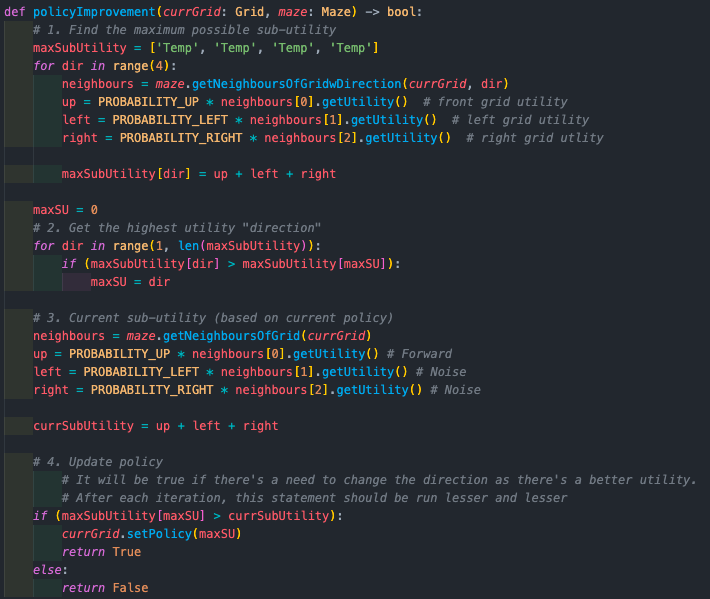


Figure 2.3: “Policy Improvement” algorithm

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

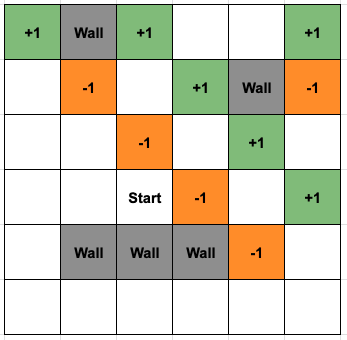
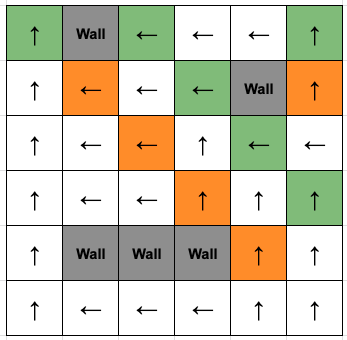
 

Figure 2.4: Plot of optimal policy for policy interaction with K = 100; 6 Iterations

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

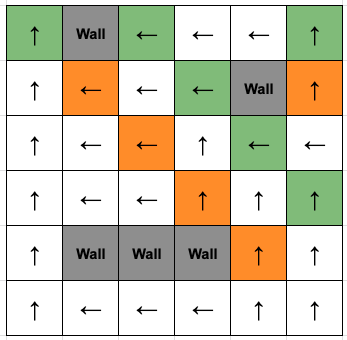
 

Figure 2.5: Utilities for policy interaction with K = 100; 6 Iterations

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

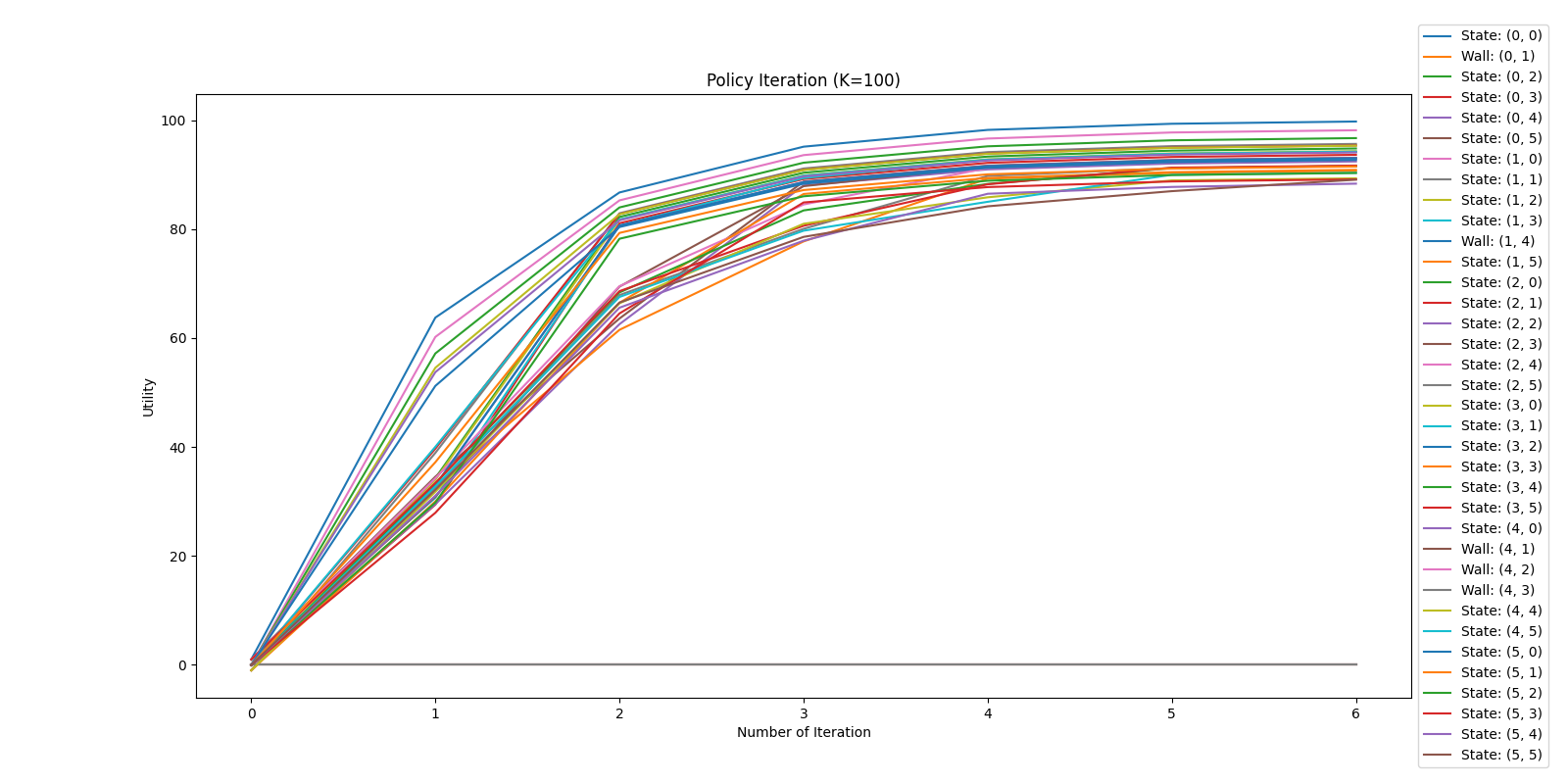


Figure 2.6: Plot of utility estimates against number of iterations with K = 100

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

## 

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

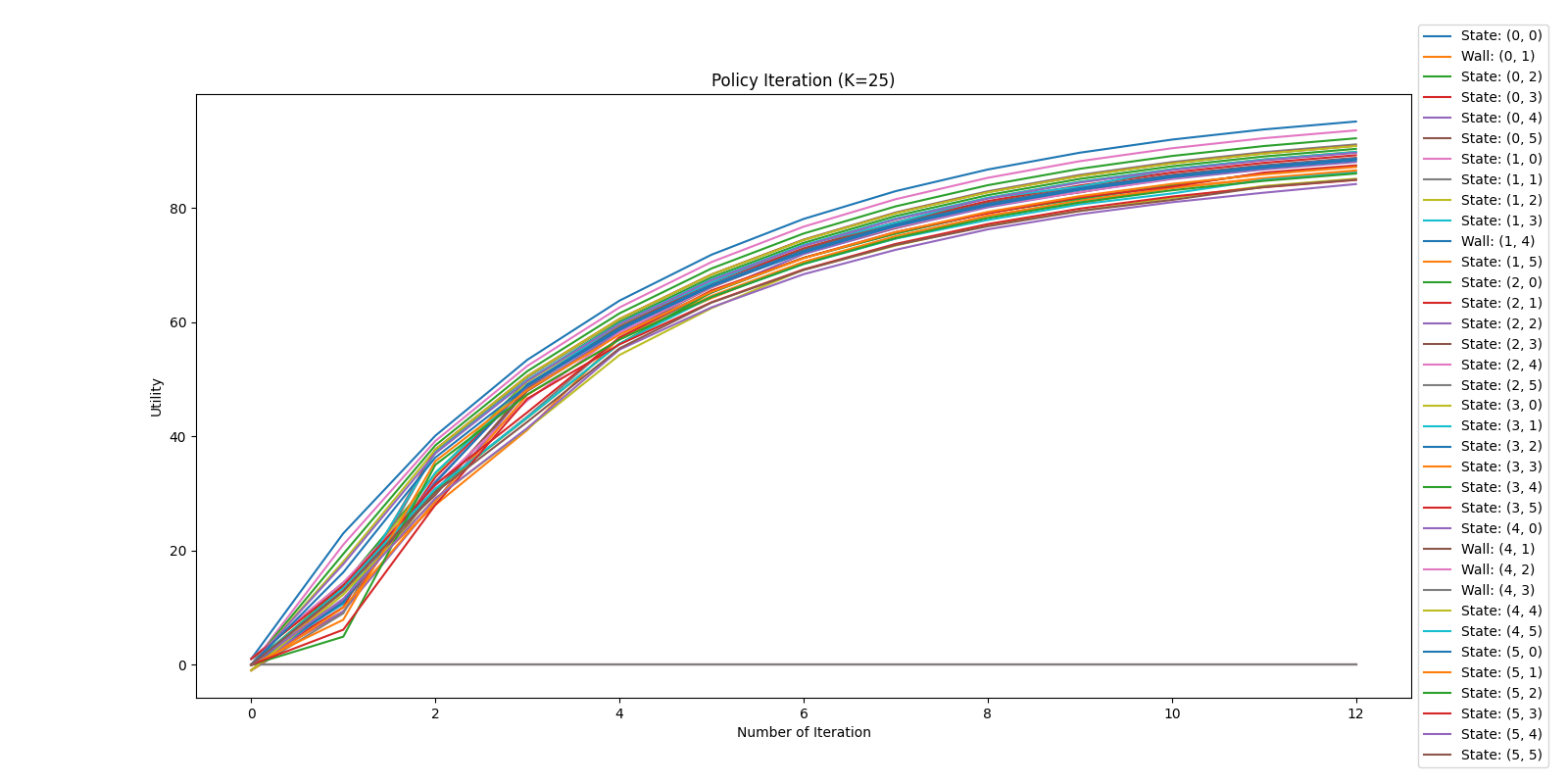


Figure 2.8: Plot of utility estimates against number of iterations with K = 25

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.

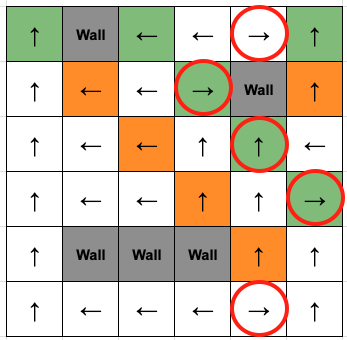
 

Figure 2.9: Utilities and Policies for policy interaction with K = 5; 10 Iterations

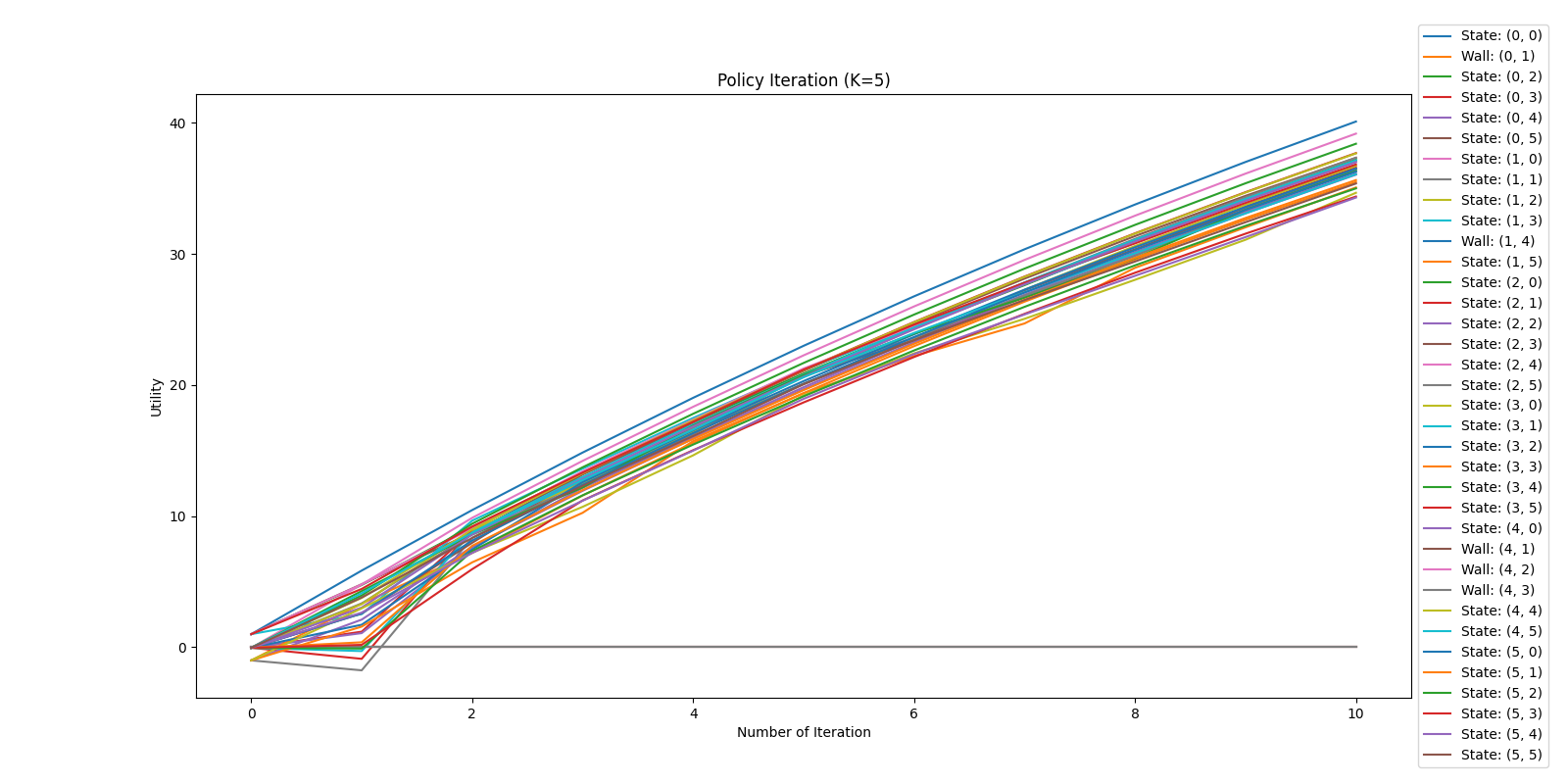


Figure 2.10: Plot of utility estimates against number of iterations with K = 5

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

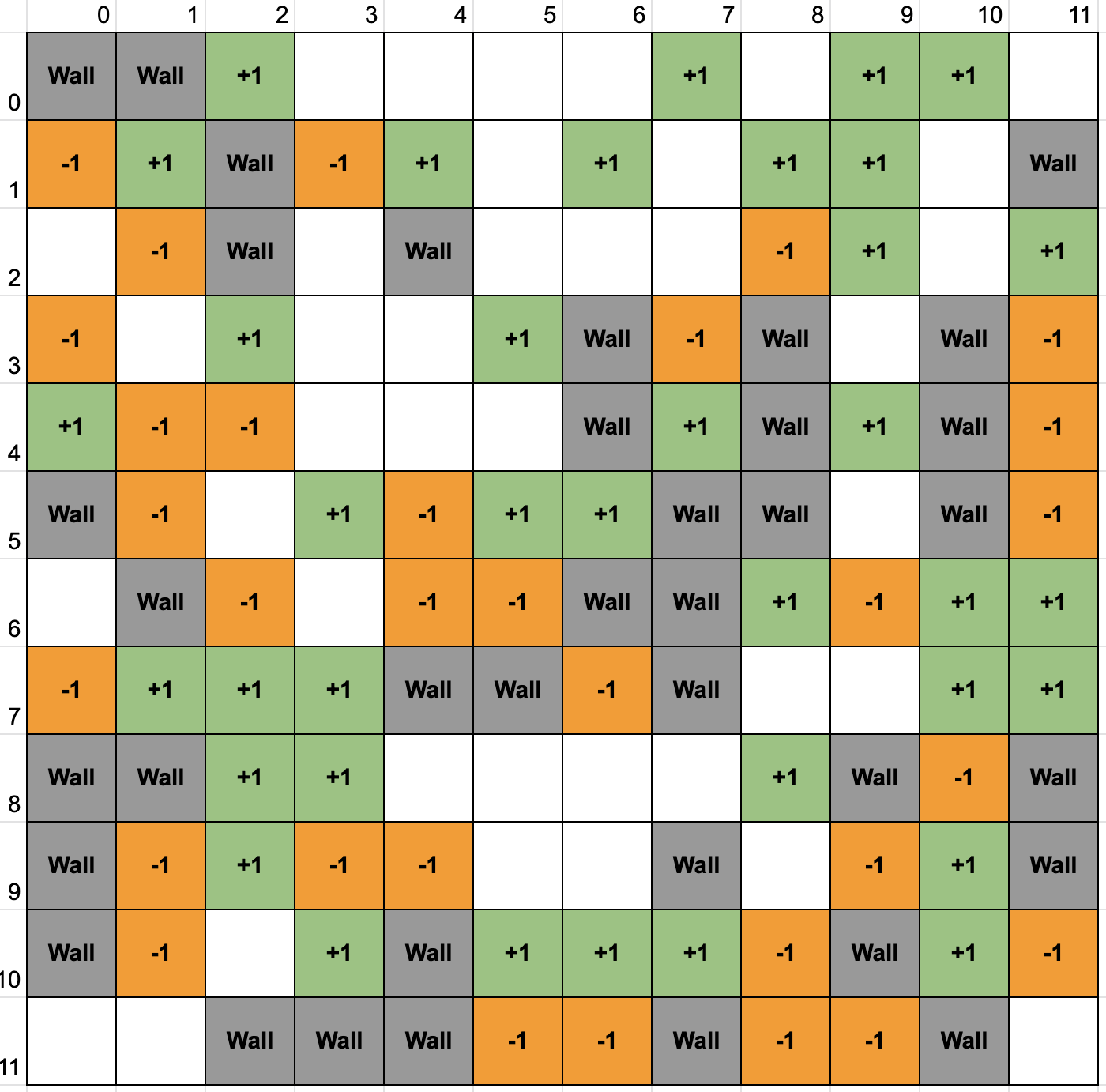


Figure 4.1: Complex Maze generated using randomMap.py

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

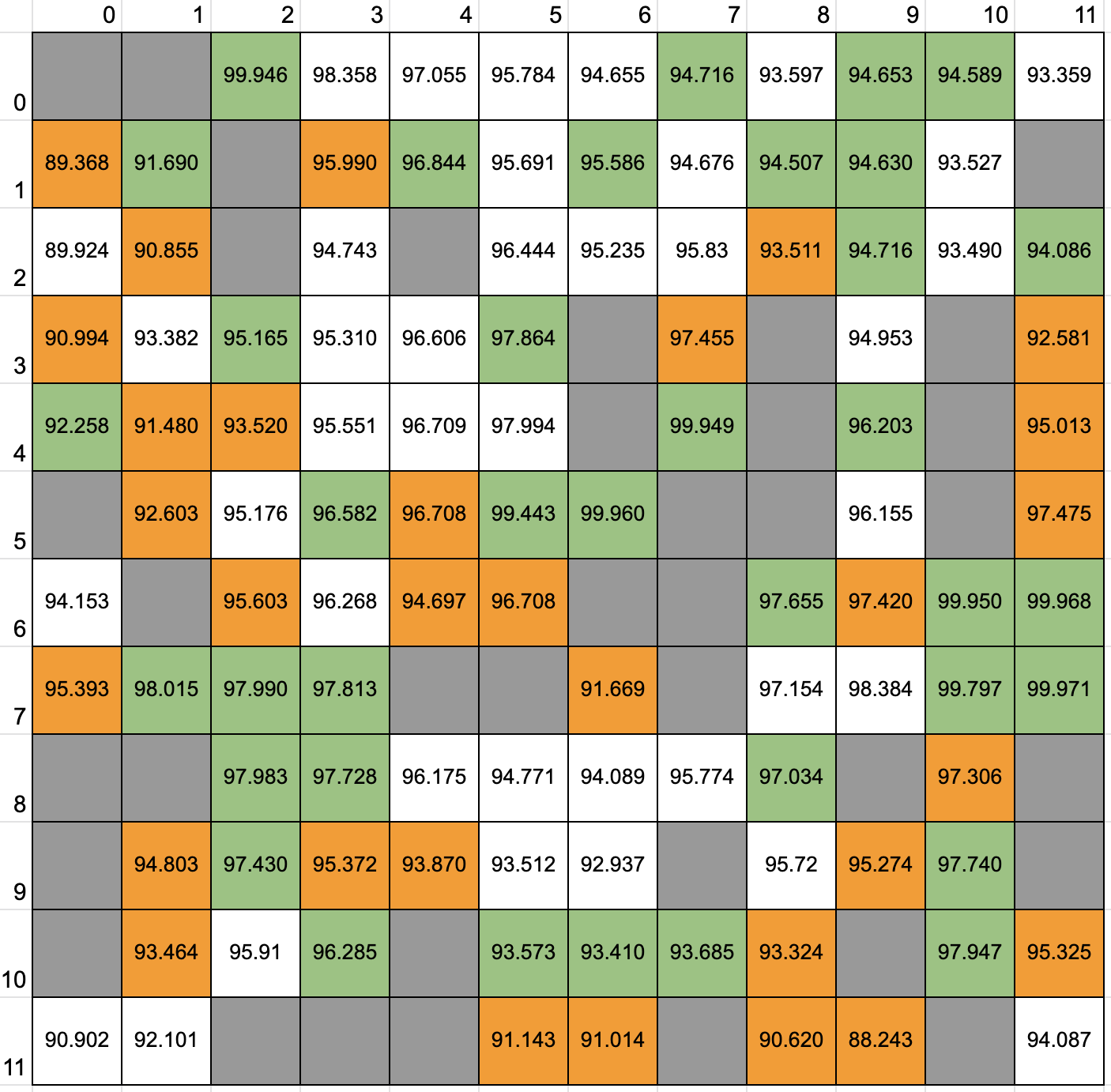
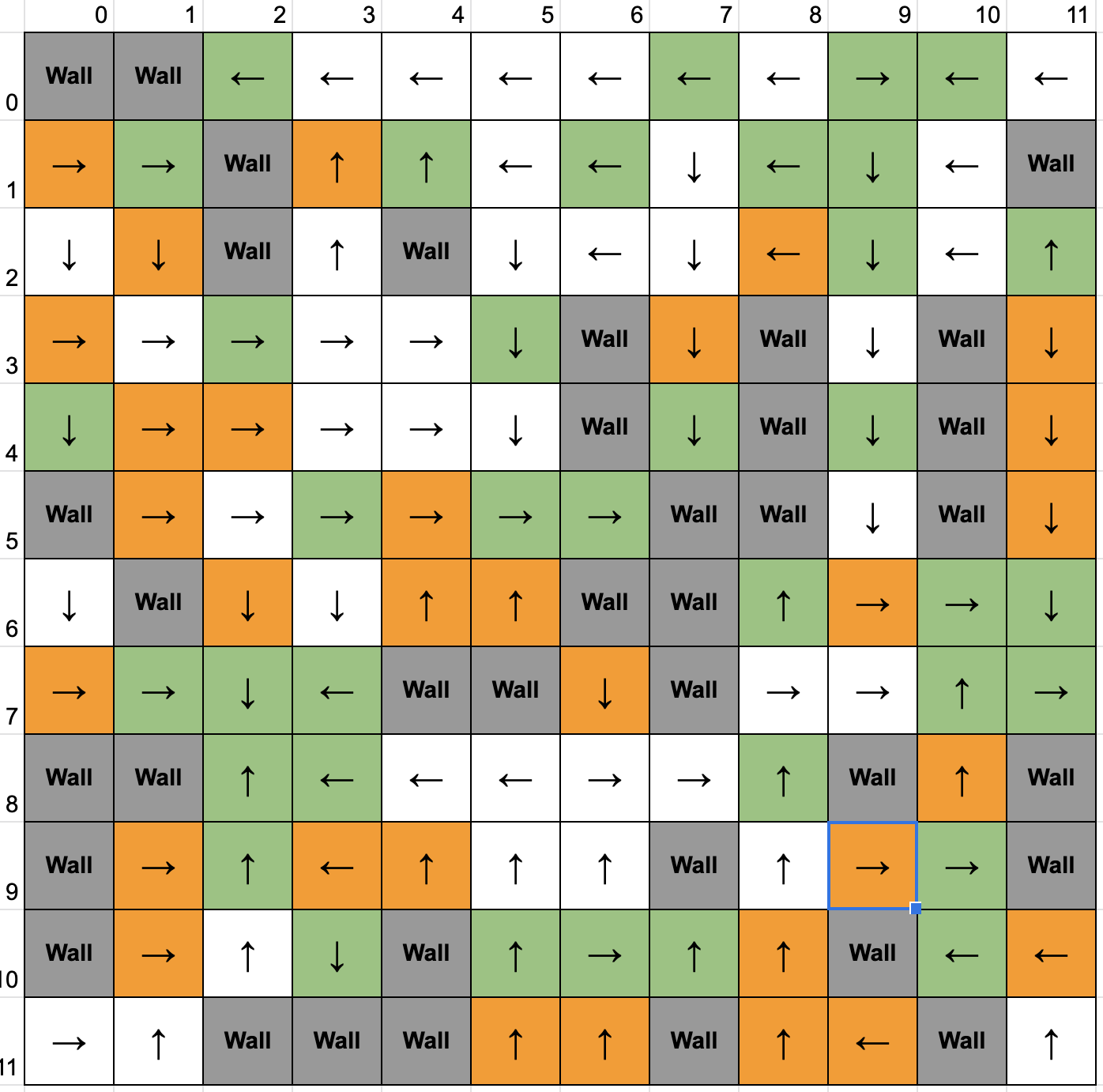


Figure 4.1: Value interaction with C = 0.01; 718 Iterations

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

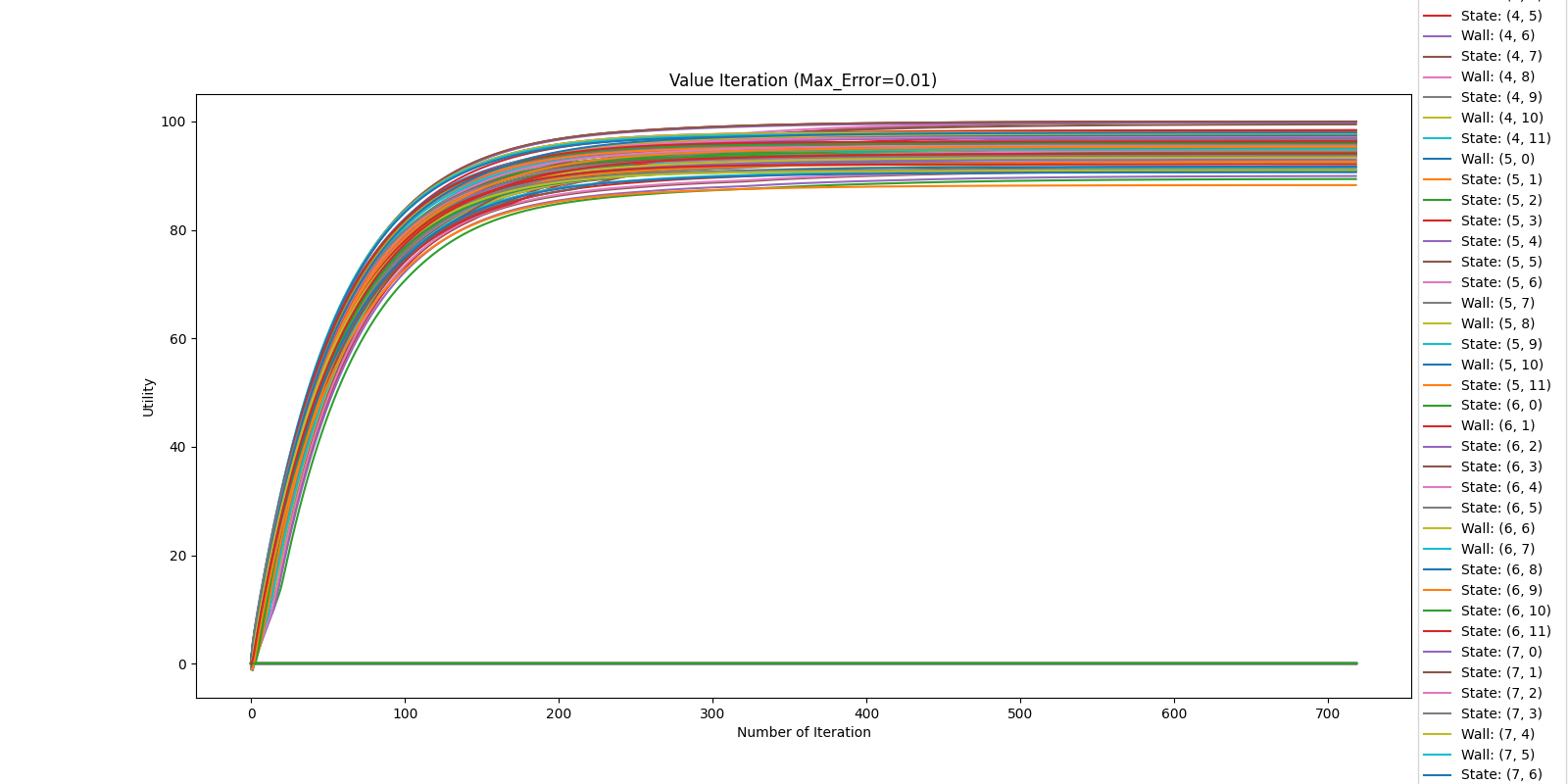


Figure 4.2: Plot of utility estimates against number of iterations with C = 0.01

## 2. Policy iteration

### 2.1 Plot of optimal policy and utilities

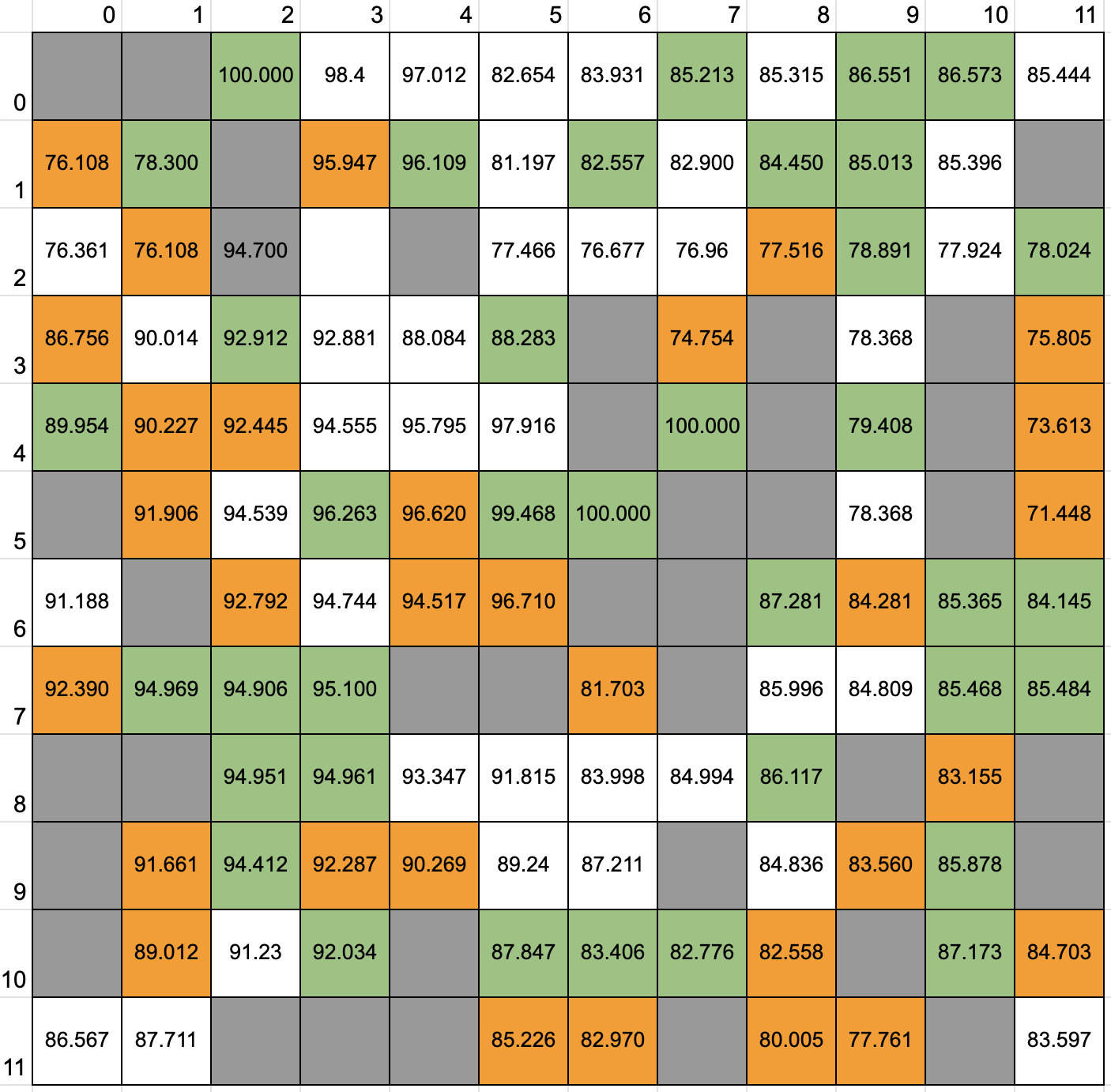
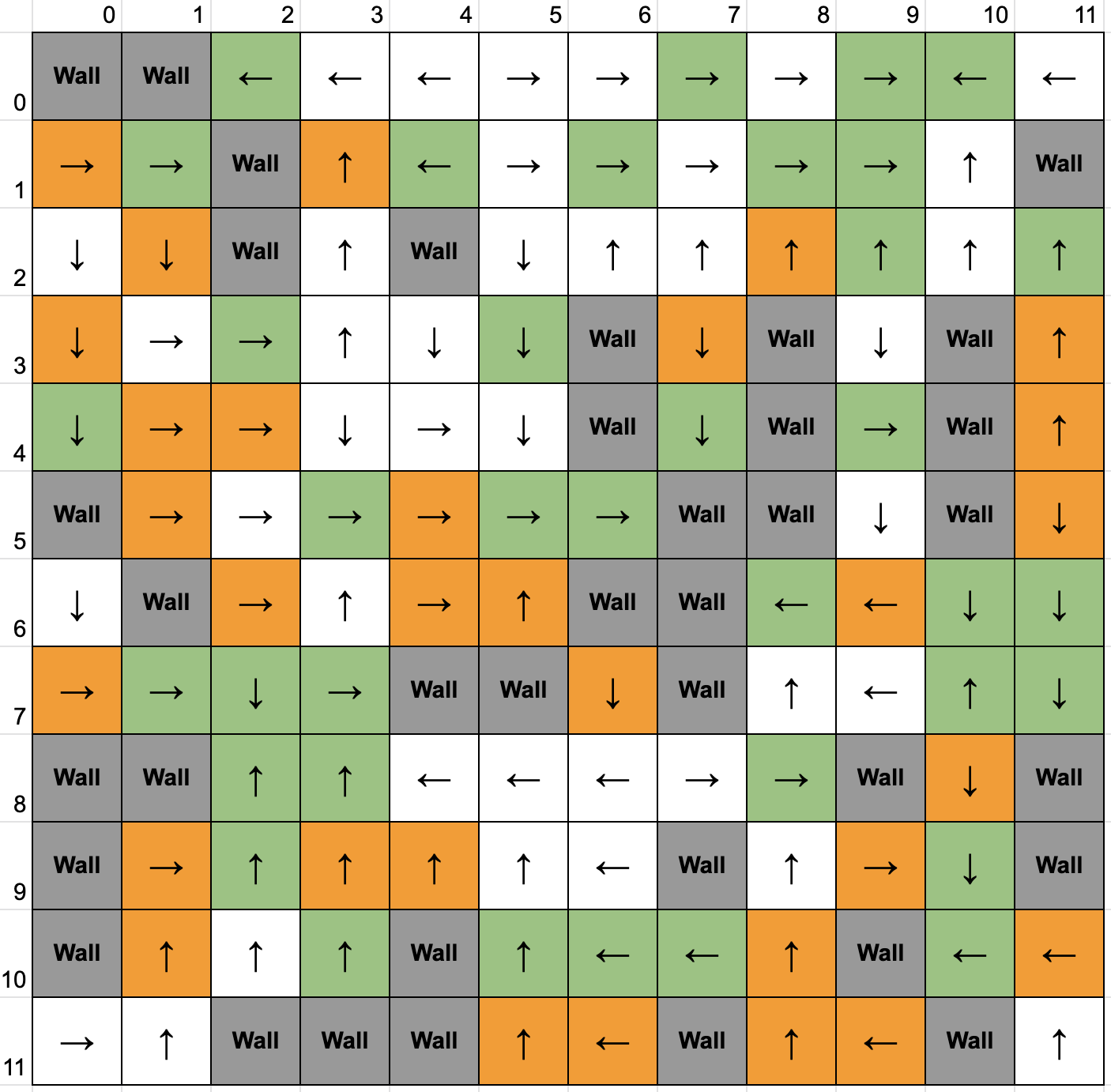


Figure 4.3: Value interaction with K = 10000; 2 Iterations

### 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

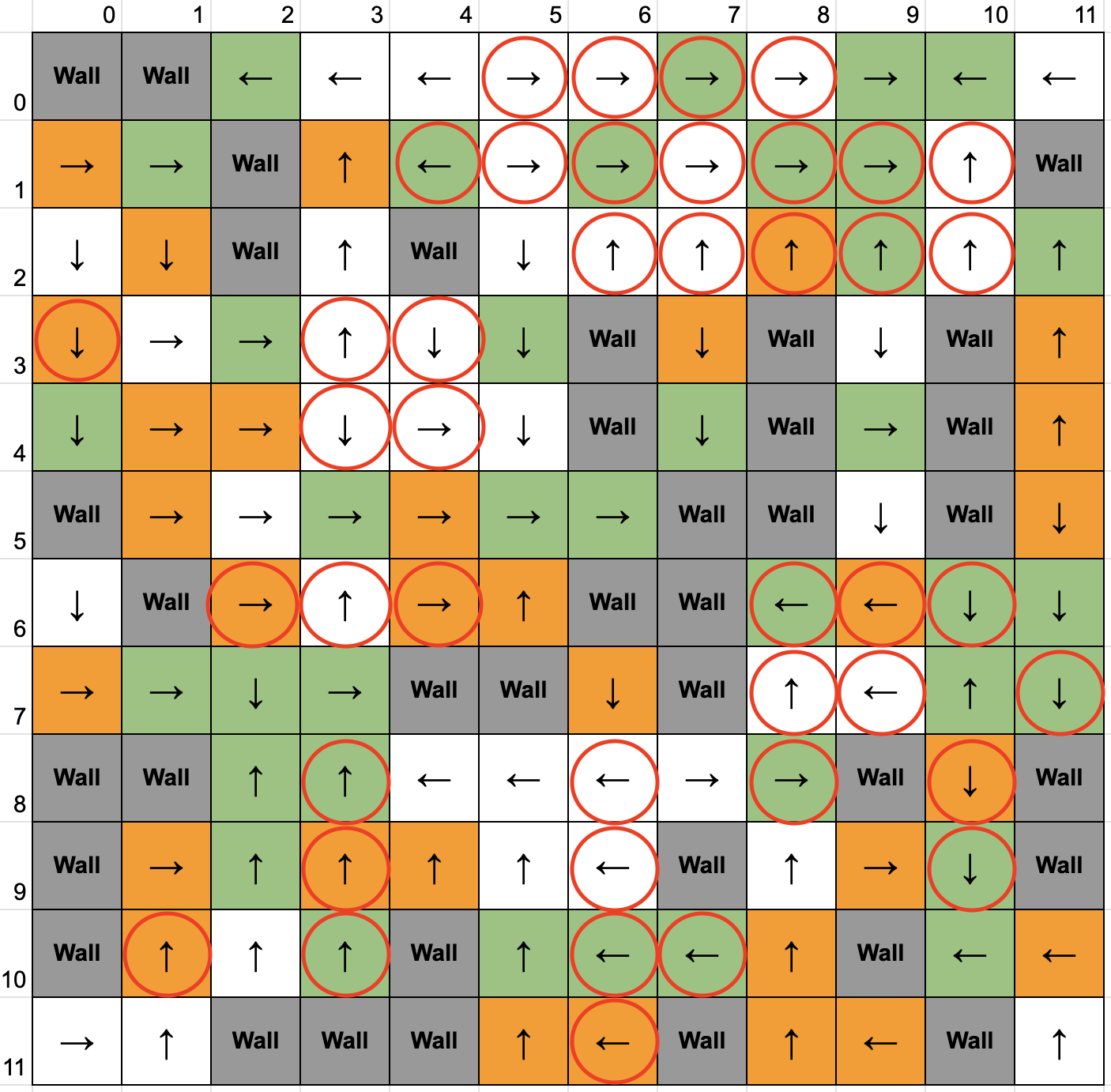
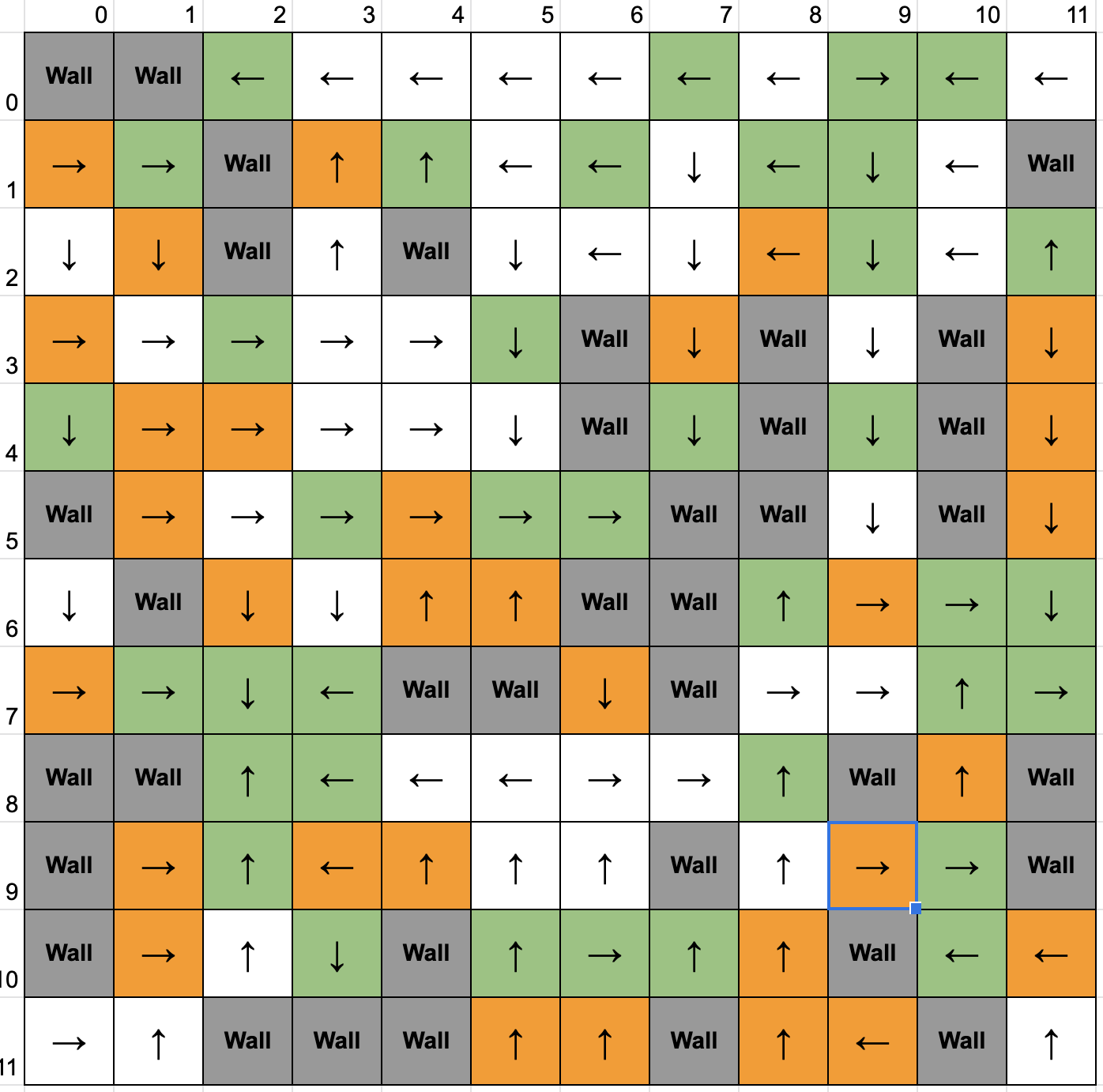


Figure 4.7: Juxtaposition of Policy between Value iteration and Policy Iteration

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.