Nanyang Technological University

College of Computing and Data Science

SC4003 CE/CZ4046 INTELLIGENT AGENTS

Assignment 1

Name	Matric No.
Lian Hong Shen Jordan	U2122011E

Contents

Description of Solution

The report will cover the solution to solve a maze environment, like the one in Section 17.1 of the reference book “Artificial Intelligence: A Modern Approach” by S. Russell and P. Norvig. Prentice-Hall, third edition, 2010. The problem that we will solve is a sequential decision problem for a fully observable stochastic environment, which the transition model has a Markovian property, and additive rewards is also known as Markov Decision Process. The approach that we are using is the Value Iteration and Policy Iteration algorithms to solve this issue. Below shows the maze environment and the transition model.

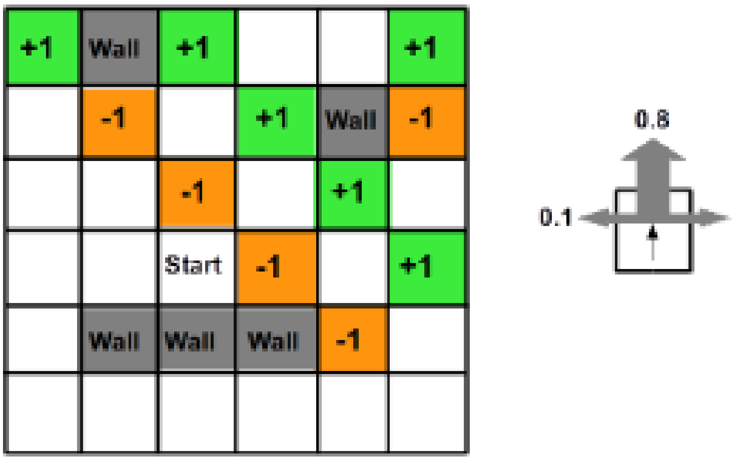


Figure 1. Maze Environment and Transition Model

The setting up of the environment along with the solution to the problem was implemented in python. To run the code:

1. Ensure python is installed, either from the official installer https://www.python.org/downloads/, or from a package manager. The python version I used was version 3.13.2.
2. Either create a virtual environment and activate it, or just use the default global environment
3. Install the packages required to run the code by navigating to the folder and running “pip install -r requirements.txt”
4. Finally, use python to execute the **main.py** file. Any constants for values of “c” and “k” can be altered at the top of the program in **main.py**. The PART1_C_VALUE changes the optimal value of c we will use for part 1, PART2_C_VALUE changes the optimal value of c we will use for part 2 and PART2_K_VALUE changes the optimal value of k we will use for part 2.

I will briefly describe what each of the files do along with their functions:

- **main.py**: This file is the entry point for the program.
  - Defines the grid environments for both Part 1 and Part 2.
  - Sets up parameters like discount factor, rewards, and action probabilities.
  - Executes both Value Iteration and Policy Iteration algorithms.
  - Compares the resulting policies for validation.
- **libraries/action.py**: This file defines the core movement of the agent in the environment.
  - **move**: returns the position of the agent after taking an action, checking for wall boundaries and collisions.
  - **side_actions**: returns the possible side actions that might occur due to the stochastic nature of the environment.

- **libraries/value_iteration.py**: This file implements the value iteration algorithm.
  - **value_iteration**: iteratively applies the Bellman update to calculate the expected utilities of states until convergence is reached based on a threshold parameter.
- **libraries/policy_iteration.py**: This file contains implementation of both Regular Policy Iteration and Modified Policy Iteration, the former using a threshold, the latter using k iterations.
  - **policy_evaluation**: calculates the utility of states under a fixed policy.
  - **policy_iteration**: alternates between policy evaluation and policy improvement until convergence.
  - **policy_evaluation_modified**: like **policy_evaluation**, but runs for exactly k iterations instead of until convergence.
  - **policy_iteration_modified**: like policy_iteration, but uses the k-iteration evaluation approach rather than threshold-based convergence.
- **libraries/utilities.py**: This file provides visualization and utility functions.
  - **plot_utilities**: generates plots showing the utility values over iterations.
  - **visualize_policy**: displays the optimal policy using directional arrows.
  - **save_utilities**: outputs utility values in readable formats.
  - **visualize_grid**: displays the grid environment with colours.
- **libraries/algorithm_evaluations.py**: This file contains functions for experimental analysis.
  - **Part1_VI_different_c_values**: analyses how different values of c affect Value Iteration.
  - **Part1_PI_different_c_values**: analyses how different values of c affect Policy Iteration.
  - **Part2_VI_different_c_values**: analyses how different values of c affect Value Iteration on the complex environment.
  - **Part2_PI_PIModified_different_k_values**: compares Regular Policy Iteration with Modified Policy Iteration using different k values on a complex grid environment.

Part 1:

In Part 1, we implement and analyse both the Value Iteration and Policy Iteration algorithms to find the optimal policy for the 6x6 grid environment defined earlier.

Environment Set up

We define the environment in **main.py**, where it's represented by a 2D list. The rewards are represented by a dictionary for quick look up of a reward a grid is supposed to give. Each element in the grid is represented by a specific state type with these representations where:

- 'G' for green cells (reward +1)
- 'B' for brown cells (reward -1)
- 'WHITE' for white cells (reward -0.05)
- 'WALL' for walls (reward 0, cannot be entered)

```
grid = [
    ['G', 'WALL', 'G', 'WHITE', 'WHITE', 'G'],
    ['WHITE', 'B', 'WHITE', 'G', 'WALL', 'B'],
    ['WHITE', 'WHITE', 'B', 'WHITE', 'G', 'WHITE'],
    ['WHITE', 'WHITE', 'WHITE', 'B', 'WHITE', 'G'],
    ['WHITE', 'WALL', 'WALL', 'WALL', 'B', 'WHITE'],
    ['WHITE', 'WHITE', 'WHITE', 'WHITE', 'WHITE', 'WHITE']
]
visualize_grid(grid, 'part_1_results', 'Grid Environment')
```

```
rewards = {'G': 1, 'B': -1, 'WHITE': -0.05, 'WALL': 0}
```

Figure 2. Grid and Rewards

We print out the environment along with the cells indices which can be seen below.

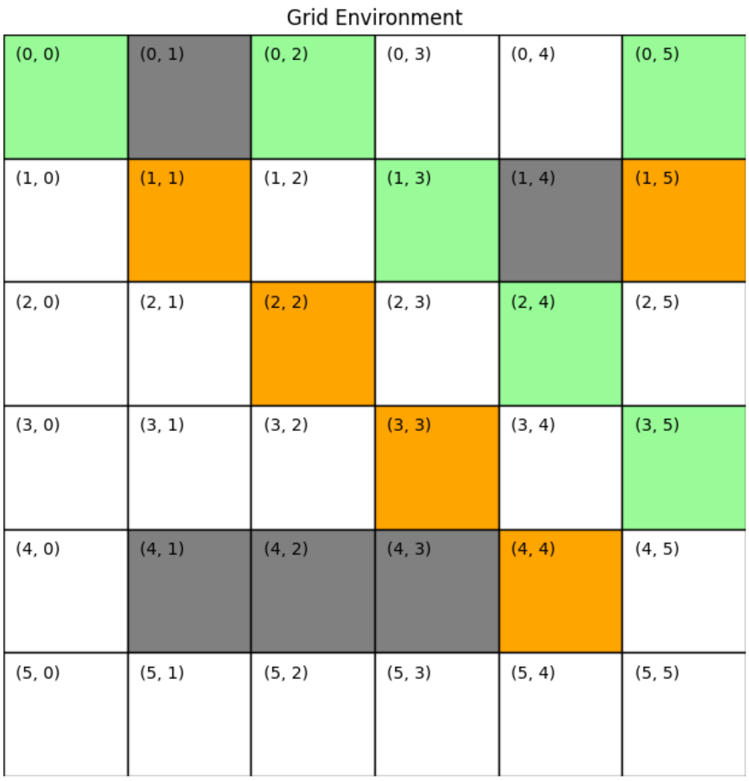


Figure 3. 6x6 Environment Grid

We also define the available actions for the actions and the transition probability using lists and dictionaries respectively. We also set the discount factor of 0.99 by assigning it to gamma.

```
actions = ['UP', 'DOWN', 'LEFT', 'RIGHT']
```

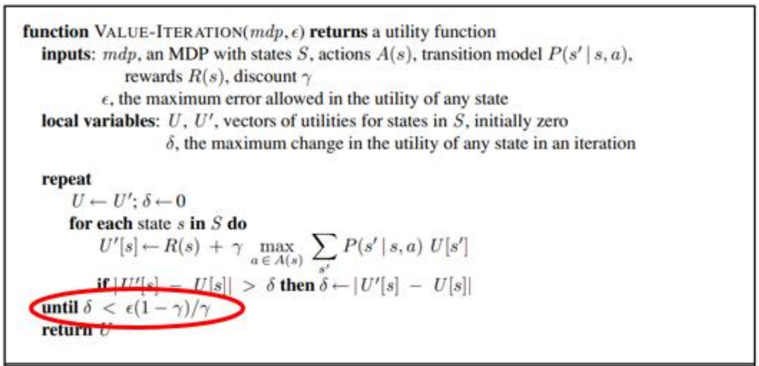
```
transition_probs = {'intended': 0.8, 'side': 0.1}
```

```
gamma = 0.99
```

Figure 4. Actions, Transition Probability and Discount Factor

Value Iteration with different c values

We first explore how the different values of the prevision parameter c will affect the Value Iteration algorithm. The c parameter is defined is the reference book (in Figure 17.5), where ε = c * Rmax, ε is the maximum error allowed in the utility of any state and Rmax is the maximum reward in the environment. We also know from the reference book (in Figure 17.4) that the termination condition for Value Iteration is given when δ < ε * (1 − γ)/γ, where delta, the Bellman error, is less than epsilon multiplied by one minus the discount factor divided by the discount factor.



function VALUE-ITERATION(mdp, ϵ) **returns** a utility function
inputs: mdp, an MDP with states S, actions $A(s)$, transition model $P(s' \mid s, a)$, rewards $R(s)$, discount γ
ϵ, the maximum error allowed in the utility of any state
local variables: U, U', vectors of utilities for states in S, initially zero
δ, the maximum change in the utility of any state in an iteration

repeat
$U \leftarrow U'$; $\delta \leftarrow 0$
for each state s **in** S **do**
$U'[s] \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\, U[s']$
if $|U'[s] - U[s]| > \delta$ **then** $\delta \leftarrow |U'[s] - U[s]|$
until $\delta < \epsilon(1-\gamma)/\gamma$
return U

Figure 17.4 The value iteration algorithm for calculating utilities of states. The termination condition is from Equation (17.8).

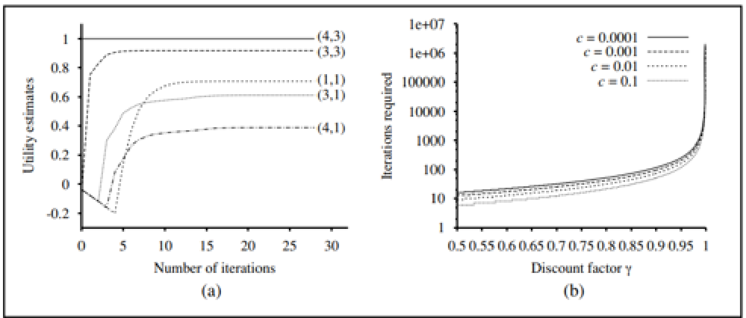


Figure 17.5 (a) Graph showing the evolution of the utilities of selected states using value iteration. (b) The number of value iterations k required to guarantee an error of at most $\epsilon = c \cdot R_{\max}$, for different values of c, as a function of the discount factor γ.

Figure 5. Figure 17.4 and Figure 17.5 from reference book

In Figure *6*, function Part1_VI_different_c_values tests the Value Iteration algorithm with four c values: 50, 10, 1, and 0.1.

```
# Precision parameter for convergence, try different c values
c_values = [50, 10, 1, 0.1]

# For Value Iteration
print("Experiment 1: Value Iteration with different values of c")
print("--------------------------------------------------------")
# COMMENT THIS LINE BELOW OUT IF YOU DON'T WANT TO RUN THE CROSS VALIDATION EXPERIMENT
Part1_VI_different_c_values(grid, rewards, actions, transition_probs, gamma, c_values, results_dir+'/exp1')
```

Figure 6. Different c values and Part1_VI_different_c_values function

Part1_VI_different_c_values function will take each value of c, and compute the threshold using the formula ε * $(1-\gamma)/\gamma$ as defined above see in Figure *7*, in lines 30-31.

```python
9  def Part1_VI_different_c_values(grid, rewards, actions, transition_probs, gamma, c_values, results_dir):
10     """...
22     vi_records_list = []
23     threshold_list = []
24 
25     # Create a figure for VI subplots
26     fig_vi, axs_vi = plt.subplots(2, 2, figsize=(14, 12))
27     axs_vi = axs_vi.flatten()
28 
29     for i, c in enumerate(c_values):
30         epsilon = c * max(rewards.values())
31         threshold = epsilon * (1 - gamma) / gamma
32         threshold = round(threshold, -int(np.floor(np.log10(abs(threshold)))))
33         threshold_list.append(threshold)
34         print(f"Running Value Iteration with c={c} (threshold={threshold})...")
35 
36         # Run Value Iteration
37         vi_utilities, vi_policy, vi_records = value_iteration(
38             grid, rewards, actions, transition_probs, gamma, threshold)
39 
40         vi_records_list.append(vi_records)
41 
42         # Plot policy in subplot
43         visualize_policy_subplot(grid, vi_policy, vi_utilities, axs_vi[i], f"c={c}, threshold={threshold}")
```

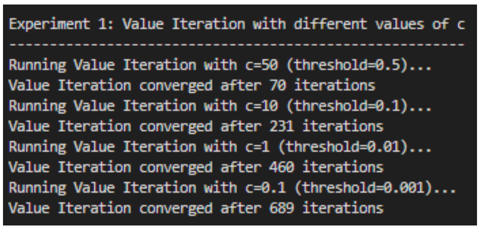
Figure 7. Part1_VI_different_c_values implementation.

We then pass the threshold the defined value_iteration function in Figure 8, run the Value Iteration algorithm until convergence. The Value Iteration algorithm initializes utility values to zero for all states in line 23, and tracks utility records for non-wall position to visualize convergence in lines 26-31. At each iteration, we apply the Bellman update to calculate new utility estimates in line 59. We continue until the maximum difference between successive utility estimates is less than the threshold seen in line 66.

```python
5  def value_iteration(grid, rewards, actions, transition_probs, gamma, threshold):
6      """...
22     height, width = len(grid), len(grid[0])
23     utilities = np.zeros((height, width))
24     policy = {}
25     # Track utility values for all non-wall positions
26     positions_to_track = []
27     for row in range(height):
28         for col in range(width):
29             if grid[row][col] != 'WALL':
30                 positions_to_track.append((row, col))
31     utility_records = {pos: [] for pos in positions_to_track}
32     iteration = 0
33     while True:
34         iteration += 1
35         delta = 0
36         new_utilities = np.zeros((height, width))
37         # Record utilities for tracked positions
38         for pos in utility_records:
39             if 0 <= pos[0] < height and 0 <= pos[1] < width:
40                 utility_records[pos].append(utilities[pos[0]][pos[1]])
41         for row in range(height):
42             for col in range(width):
43                 if grid[row][col] == 'WALL':
44                     continue
45                 # Calculate the expected utility for each action
46                 action_utilities = []
47                 for action in actions:
48                     expected_utility = 0
49                     # Calculate for intended action (probability 0.8)
50                     new_row, new_col = move(row, col, action, grid)
51                     expected_utility += transition_probs['intended'] * utilities[new_row][new_col]
52                     # Calculate for side actions (probability 0.1 each)
53                     for side_action in side_actions(action):
54                         new_row, new_col = move(row, col, side_action, grid)
55                         expected_utility += transition_probs['side'] * utilities[new_row][new_col]
56                     action_utilities.append(expected_utility)
57                 # Choose the action that maximizes utility
58                 best_action_idx = np.argmax(action_utilities)
59                 new_utilities[row][col] = rewards[grid[row][col]] + gamma * action_utilities[best_action_idx]
60                 policy[(row, col)] = actions[best_action_idx]
61                 # Update delta for convergence check
62                 delta = max(delta, abs(new_utilities[row][col] - utilities[row][col]))
63         # Update utilities for next iteration
64         utilities = new_utilities
65         # Check for convergence
66         if delta < threshold:
67             break
68     print(f"Value Iteration converged after {iteration} iterations")
69     return utilities, policy, utility_records
```

Figure 8. Implementation of Value Iteration algorithm.

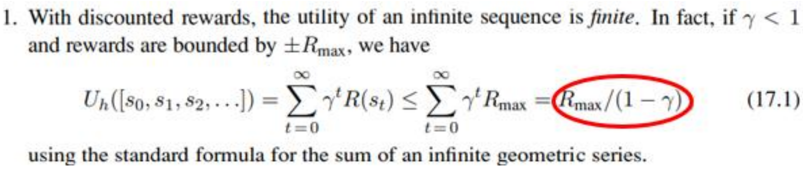
This signifies only one run for a certain c value. We collect the utilities, optimal policies and utility records at each non-wall positions for each c value and plot them. There are two main plots, utility estimates as a function of the number of iterations and the plot of optimal policy. The terminal output shows how c affects the number of iterations required for convergence as seen in Figure 9.



```
Experiment 1: Value Iteration with different values of c
--------------------------------------------------------
Running Value Iteration with c=50 (threshold=0.5)...
Value Iteration converged after 70 iterations
Running Value Iteration with c=10 (threshold=0.1)...
Value Iteration converged after 231 iterations
Running Value Iteration with c=1 (threshold=0.01)...
Value Iteration converged after 460 iterations
Running Value Iteration with c=0.1 (threshold=0.001)...
Value Iteration converged after 689 iterations
```

Figure 9. Terminal output for Experiment 1

In Figure 11, we can observe that utility values for non-wall positions converge more smoothly when the value of c is higher. The number of iterations that are required for convergence are higher but the final utility values become more accurate. The maximum utility tallies with what we know from the reference book in (17.1) as seen in Figure 10, where the utility of an infinite sequence is bounded by Rmax/(1-γ) when γ < 1. In our environment with Rmax=1 and γ=0.99, this upper bound is 100. We observe in the plots of Figure 11 that the utilities of states near reward cells approach but remain below this theoretical maximum, with the highest values being for states adjacent to +1 reward cells, where the agent has a high probability of collecting the positive reward in subsequent steps.



1. With discounted rewards, the utility of an infinite sequence is *finite*. In fact, if $\gamma < 1$ and rewards are bounded by $\pm R_{\max}$, we have

$$U_h([s_0, s_1, s_2, \ldots]) = \sum_{t=0}^{\infty} \gamma^t R(s_t) \leq \sum_{t=0}^{\infty} \gamma^t R_{\max} = R_{\max}/(1-\gamma) \qquad (17.1)$$

using the standard formula for the sum of an infinite geometric series.

Figure 10. Reference book (17.1)

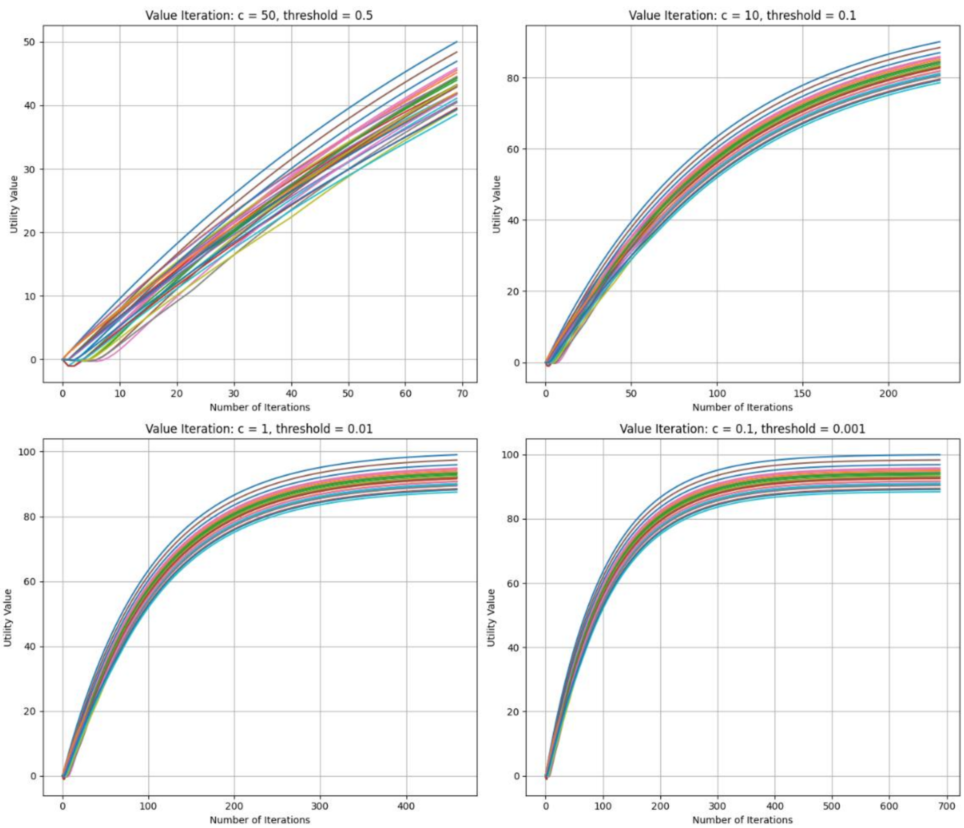


Figure 11. Plots of number of iterations against utility value for non-wall position where c = [50,10,1,0.1]

In Figure 12, we note that only for c=50 the optimal policy is different at (4,5) where instead of "LEFT" it is "RIGHT". This might be due to premature convergence caused by the high threshold value (0.5), which stops the value iteration process before the utilities have fully stabilized. With a larger threshold, the algorithm terminates before it can accurately determine the subtle differences in expected utility that would lead to choosing "LEFT" as the optimal action at this position. The other three values of c (10, 1, and 0.1) all converge to the same optimal policy, suggesting that c=10 is sufficient for finding the correct policy for this environment, while c=50 leads to a suboptimal solution at certain states due to its early termination.

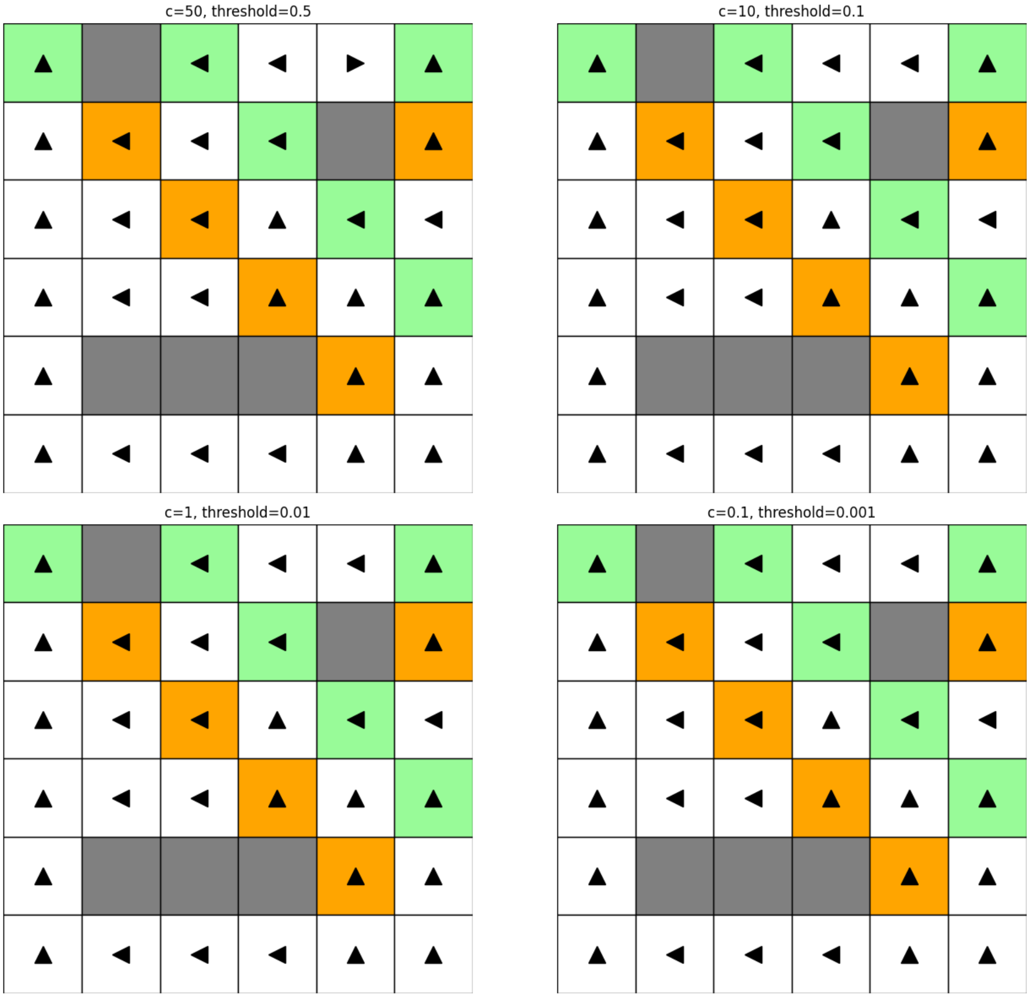


Figure 12. Plot of Optimal Policy Grids where c = [50,10,1,0.1]

These results confirm the theoretical relationship between the threshold value and convergence speed: a higher threshold (larger c) leads to faster convergence but potentially less precise results, while a lower threshold (smaller c) requires more iterations but produces more accurate utility values.

Results of Value Iteration with c=1 (threshold=0.01)

Based on the experimental results, I selected c=1 as a good balance between computational efficiency and solution accuracy, giving a threshold of 0.01. With this threshold, Value Iteration converged after 460 iterations. With a discount factor of γ=0.99, the theoretical maximum utility for an infinite sequence is Rmax/(1-γ) = 1/(1-0.99) = 100, which is what we observe the utility of cell (0,0) approaches. From this position, the agent can reliably reach the +1 reward state by moving up even with the stochastic nature of the environment. There's no risk of entering the negative reward state.

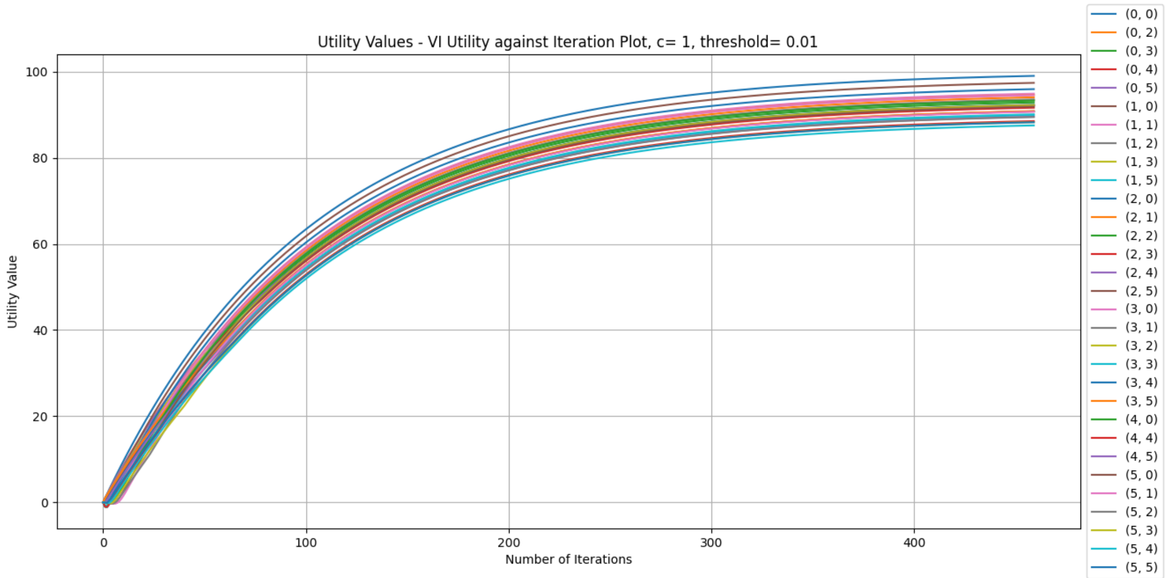


Figure 13. Value Iteration Plot of utility estimates as a function of the number of iterations, c =1

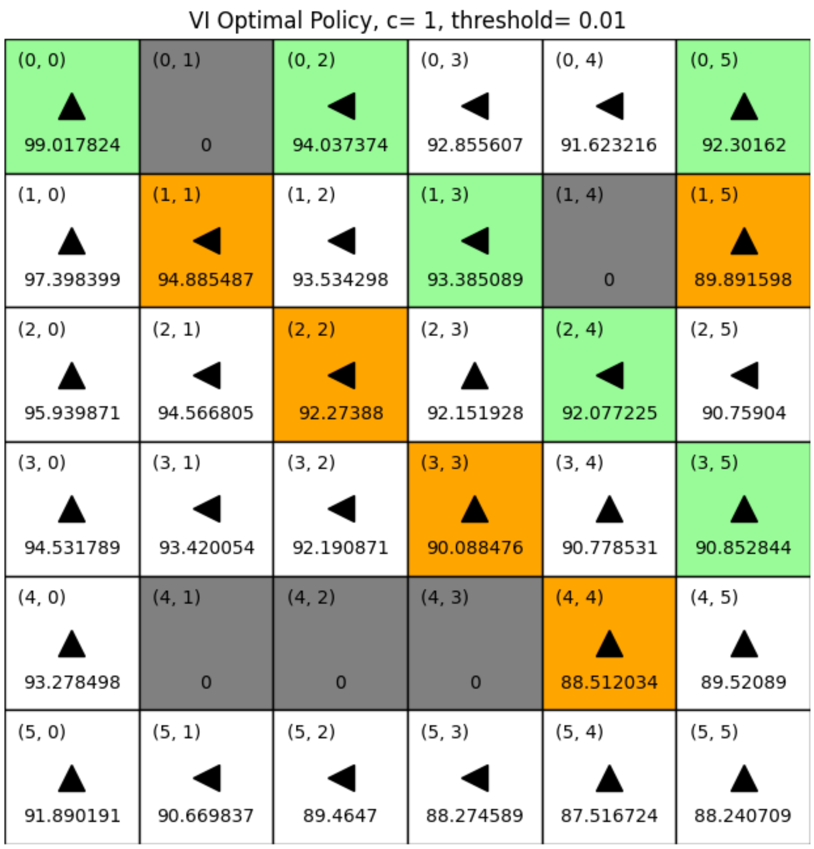


Figure 14. Plot of optimal policy and Utilities of all states, c = 1

UTILITY VALUE OF ALL STATES

	0	1	2	3	4	5
0	99.017824	0	94.037374	92.855607	91.623216	92.301620
1	97.398399	94.885487	93.534298	93.385089	0	89.891598
2	95.939871	94.566805	92.273880	92.151928	92.077225	90.759040
3	94.531789	93.420054	92.190871	90.088476	90.778531	90.852844
4	93.278498	0	0	0	88.512034	89.520890
5	91.890191	90.669837	89.464700	88.274589	87.516724	88.240709

Figure 15. Utilities of all States in markdown table form, c = 1

Policy Iteration with different c values

We run Policy Iteration algorithm with the same values of c as seen in Figure 16.

```
# For Policy Iteration
print("Experiment 2: Policy Iteration with different values of c")
print("---------------------------------------------------------")
# COMMENT THIS LINE BELOW OUT IF YOU DON'T WANT TO RUN THE CROSS VALIDATION EXPERIMENT
Part1_PI_different_c_values(grid, rewards, actions, transition_probs, gamma, c_values, results_dir+'/exp2')
```

Figure 16. Part1_PI_different_c_values function

Part1_PI_different_c_values function will take the same four values of c: 50, 10, 1, and 0.1, like our Value Iteration experiments. The threshold values calculated are identical to those used in Value Iteration. The Policy Iteration Algorithm in Figure 17, first gets a fixed initial policy by setting all optimal actions to be “UP” as seen in lines 84-87. Only when policy remains unchanged from policy improvement then we stop, seen in lines 127-128, we set policy_stable to false and break the loop.

```
64  def policy_iteration(grid, rewards, actions, transition_probs, gamma, threshold):
65      """...
82      height, width = len(grid), len(grid[0])
83      utilities = np.zeros((height, width))
84      # Initialize policy
85      policy = {}
86      for row in range(height):
87          for col in range(width):
88              if grid[row][col] != 'WALL':
89                  policy[(row, col)] = 'UP'  # Fixed initial policy
90      # Track utility values for all non-wall positions
91      positions_to_track = []
92      for row in range(height):
93          for col in range(width):
94              if grid[row][col] != 'WALL':
95                  positions_to_track.append((row, col))
96      # Modified utility records to track both evaluation iterations and policy changes
97      policy_eval_records  = {position: [] for position in positions_to_track}
98      policy_improv_records = {position: [] for position in positions_to_track}
99      policy_eval = 0
100     policy_improv = 0
101     while True:  # Policy is stable for a single iteration
102         # Policy evaluation
103         utilities, eval_iterations = policy_evaluation(policy, utilities, grid, rewards, actions, transition_probs, gamma, policy_eval_records, policy_improv_records, threshold)
104         policy_eval += eval_iterations
105         # Policy improvement
106         policy_stable = True
107         policy_improv += 1
108         for row in range(height):
109             for col in range(width):
110                 if grid[row][col] == 'WALL':
111                     continue
112                 old_action = policy.get((row, col))
113                 # Calculate expected utilities for all actions
114                 action_utilities = []
115                 for action in actions:
116                     expected_utility = 0
117                     # For intended action
118                     new_row, new_col = move(row, col, action, grid)
119                     expected_utility += transition_probs['intended'] * utilities[new_row][new_col]
120                     # For side actions
121                     for side_action in side_actions(action):
122                         new_row, new_col = move(row, col, side_action, grid)
123                         expected_utility += transition_probs['side'] * utilities[new_row][new_col]
124                     action_utilities.append(expected_utility)
125                 # Choose the best action
126                 best_action_idx = np.argmax(action_utilities)
127                 policy[(row, col)] = actions[best_action_idx]
128                 # Check if policy changed
129                 if old_action != policy[(row, col)]:
130                     policy_stable = False
131         # Policy is stable for a single iteration , we've found the optimal policy
132         if policy_stable:
133             break
134     print(f"Policy Iteration converged after {policy_eval} policy evaluation iterations and {policy_improv} policy improvement iterations")
135     return utilities, policy, policy_eval_records, policy_improv_records
```

Figure 17. Implementation of Policy Iteration algorithm.

For policy evaluation, see Figure 18, where we track the number of iterations in the policy evaluation step seen in line 33. We continue the policy evaluation until the delta is less than threshold, in line 59. We also track the number of iterations in the policy improvement step seen in Figure 17, line 107.

```
6  def policy_evaluation(policy, utilities, grid, rewards, actions, transition_probs, gamma, policy_eval_records, policy_improv_records, threshold):
7      """...
26     height, width = len(grid), len(grid[0])
27     # Record policy improvement utilities for tracked positions
28     for pos in policy_improv_records:
29         if 0 <= pos[0] < height and 0 <= pos[1] < width:
30             policy_improv_records[pos].append(utilities[pos[0]][pos[1]])
31     eval_iterations = 0
32     while True:
33         eval_iterations += 1
34         delta = 0
35         new_utilities = np.zeros((height, width))
36         # Record policy evaluation utilities for tracked positions
37         for pos in policy_eval_records:
38             if 0 <= pos[0] < height and 0 <= pos[1] < width:
39                 policy_eval_records[pos].append(utilities[pos[0]][pos[1]])
40         for row in range(height):
41             for col in range(width):
42                 if grid[row][col] == 'WALL':
43                     continue
44                 # Default to UP if no policy defined
45                 action = policy.get((row, col), 'UP')
46                 # Calculate expected utility
47                 expected_utility = 0
48                 # For intended action
49                 new_row, new_col = move(row, col, action, grid)
50                 expected_utility += transition_probs['intended'] * utilities[new_row][new_col]
51                 # For side actions
52                 for side_action in side_actions(action):
53                     new_row, new_col = move(row, col, side_action, grid)
54                     expected_utility += transition_probs['side'] * utilities[new_row][new_col]
55                 new_utilities[row][col] = rewards[grid[row][col]] + gamma * expected_utility
56                 delta = max(delta, abs(new_utilities[row][col] - utilities[row][col]))
57         utilities = new_utilities
58         # Check for convergence
59         if delta < threshold:
60             break
61     return utilities, eval_iterations
```

Figure 18. Implementation of Policy Evaluation algorithm.

The terminal output shows how c affects the number of iterations required for convergence as seen in Figure 19. For illustration purposes, we will use the utility value against policy evaluation iterations to have a wider range of values to compare as compared to policy improvement iterations.

```
Experiment 2: Policy Iteration with different values of c
-------------------------------------------------------
Running Policy Iteration with c=50 (threshold=0.5)...
Policy Iteration converged after 131 policy evaluation iterations and 5 policy improvement iterations
Running Policy Iteration with c=10 (threshold=0.1)...
Policy Iteration converged after 395 policy evaluation iterations and 5 policy improvement iterations
Running Policy Iteration with c=1 (threshold=0.01)...
Policy Iteration converged after 860 policy evaluation iterations and 5 policy improvement iterations
Running Policy Iteration with c=0.1 (threshold=0.001)...
Policy Iteration converged after 1342 policy evaluation iterations and 5 policy improvement iterations
```

Figure 19. Terminal output for Experiment 2

In Figure 20, shows the utility values for non-wall positions across evaluation iterations for different c values. As expected, we can see that a smaller c, means smaller threshold requires more policy evaluation iterations for convergence. Each plot shows distinct "steps" or plateaus in the utility curves, which likely correspond to the policy improvement iterations. After each policy improvement step, the utility values adjust to the new policy and then stabilize until the next policy change.

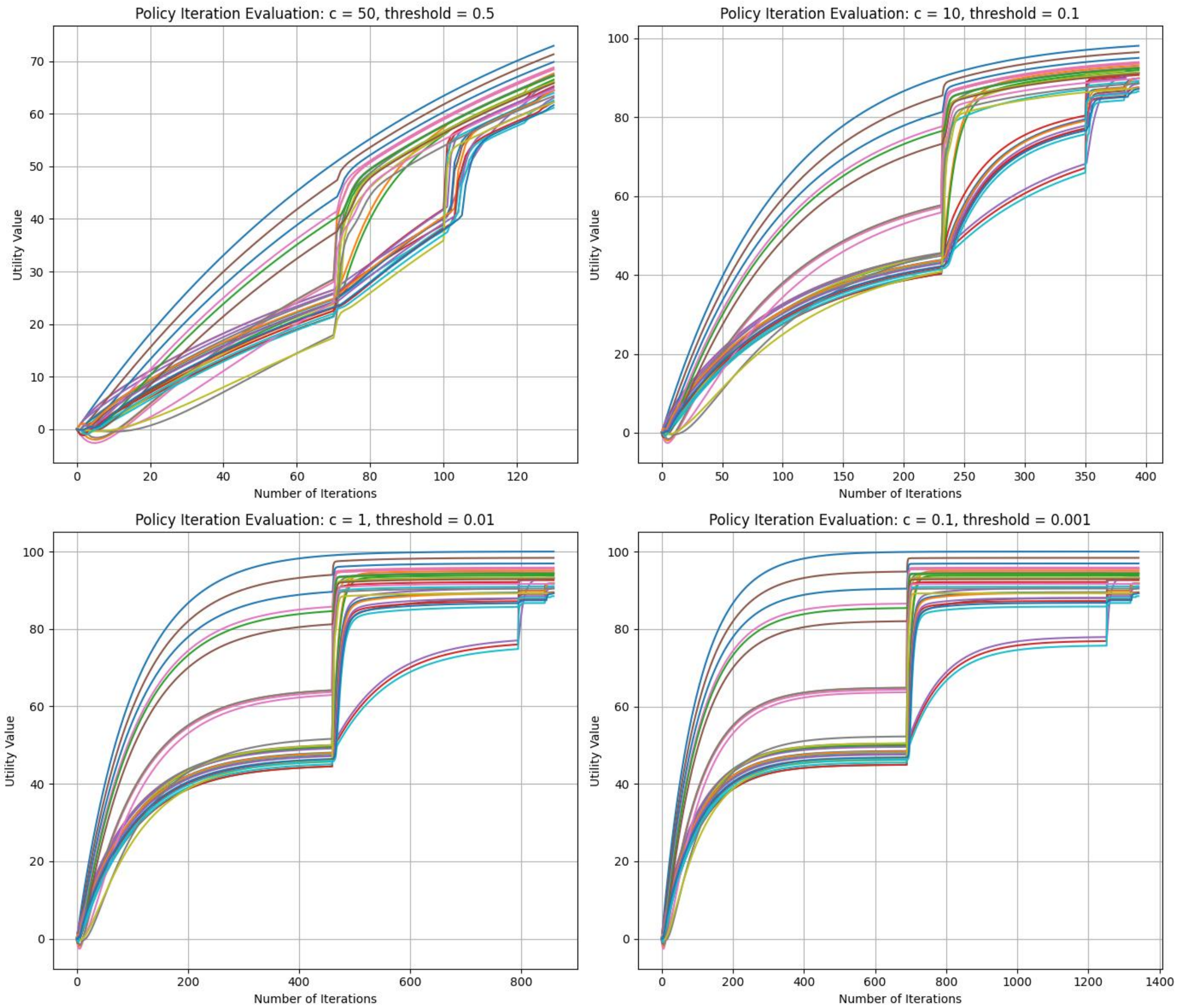


Figure 20. Plots of number of evaluation iterations against utility value for non-wall position where c = [50,10,1,0.1]

In Figure 21, we note that only for c=50 the optimal policy is different at (3,3) where instead of “UP” it is “LEFT”. This might be due to premature convergence caused by the high threshold value (0.5), which stops the policy evaluation process before the utilities have fully stabilized. With a larger threshold, the algorithm terminates before it can accurately determine the subtle differences in expected utility that would lead to choosing "UP" as the optimal action at this position. The other c values have policies that are identical, suggesting that c=10 is sufficient for finding the correct policy for this environment, while c=50 leads to a suboptimal solution at certain states due to its early termination.

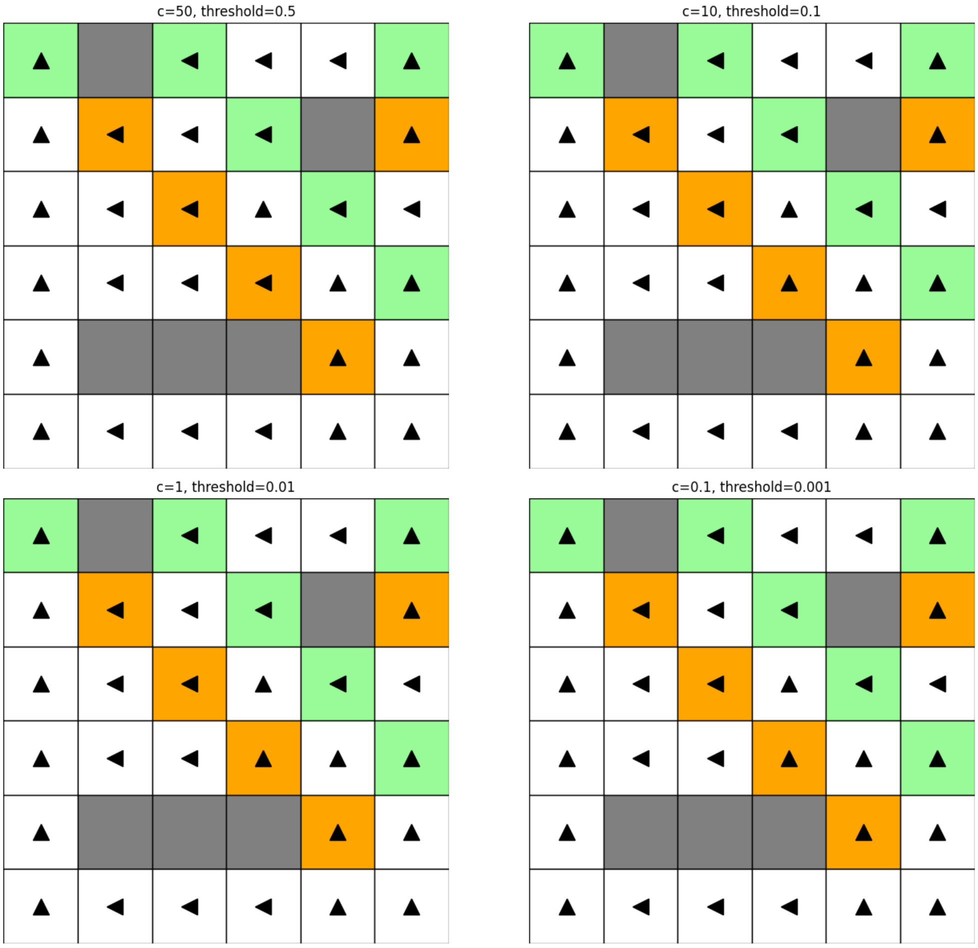


Figure 21. Plot of Optimal Policy Grids where c = [50,10,1,0.1]

Results of Policy Iteration with c=1 (threshold=0.01)

Based on the experimental results, I selected c=1 as a good balance between computational efficiency and solution accuracy, giving a threshold of 0.01. With this threshold, Policy Iteration converged after 860 policy evaluation iterations and 5 policy improvement iterations.

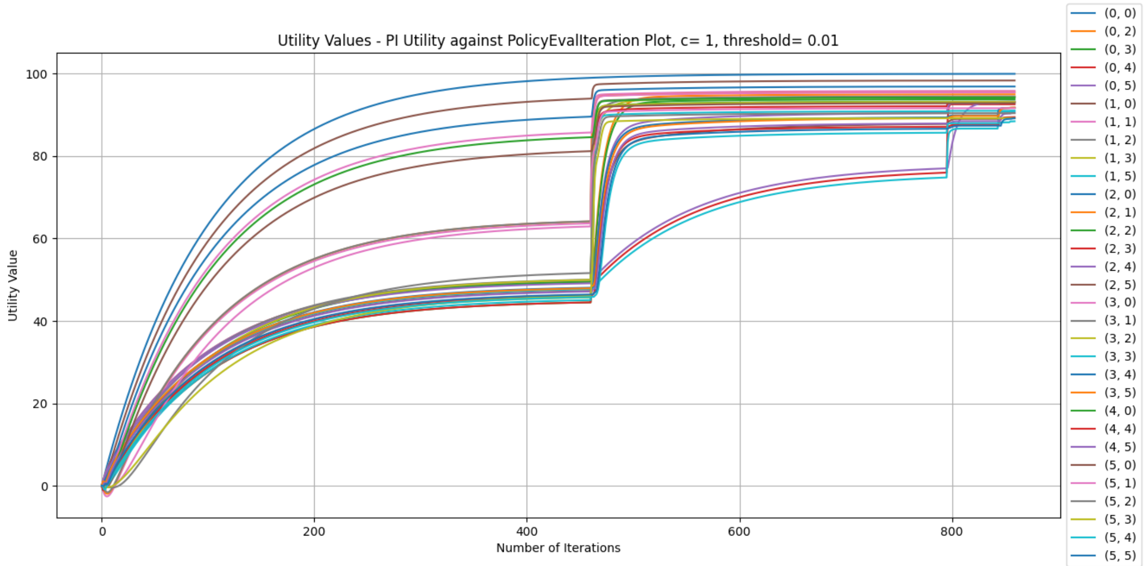


Figure 22. Policy Iteration Plot of utility estimates as a function of the number of iterations (eval iterations), c =1

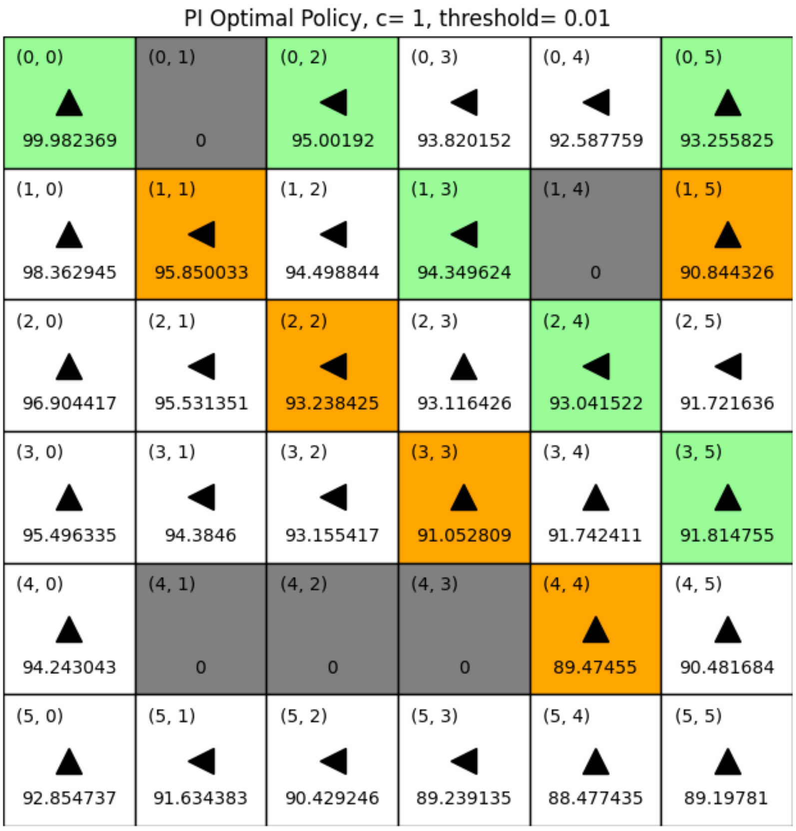


Figure 23. Plot of optimal policy and Utilities of all states, c = 1

UTILITY VALUE OF ALL STATES

	0	1	2	3	4	5
0	99.982369	0	95.001920	93.820152	92.587759	93.255825
1	98.362945	95.850033	94.498844	94.349624	0	90.844326
2	96.904417	95.531351	93.238425	93.116426	93.041522	91.721636
3	95.496335	94.384600	93.155417	91.052809	91.742411	91.814755
4	94.243043	0	0	0	89.474550	90.481684
5	92.854737	91.634383	90.429246	89.239135	88.477435	89.197810

Figure 24. Utilities of all States in markdown table form, c = 1

Comparing Optimal Policies of VI(c=1) and PI(c=1)

Lastly, to validate our results, seen in Figure 25, we compared both policies derived from Value Iteration and Policy Iteration. Both algorithms converged to identical policies for c=1, with a 100% match across all states confirming that both algorithms correctly identified the optimal policy for this environment.

```
# Validate the policies to see if they are the same
print("Comparing policies")
print("--------------------------------------------------------")
# If they are not the same, one/both of the algorithm has a bug or one/both of them have not converged
match_count = 0
total = 0
for row in range(len(grid)):
    for col in range(len(grid[0])):
        if grid[row][col] != 'WALL':
            total += 1
            if vi_policy.get((row, col)) == pi_policy.get((row, col)):
                match_count += 1

print(
    f"Policy agreement: {match_count}/{total} states ({match_count/total*100:.2f}%)\n\n")
```

```
Comparing policies
--------------------------------------------------------
Policy agreement: 31/31 states (100.00%)
```

Figure 25. Comparing optimal policies for Value Iteration and Policy Iteration, c=1

Part 2:

Environment Set up

A 12x12 grid was chosen for a more complicated maze environment. After generating a complex grid environment using a 2D list, we visualize the grid, which can be seen in Figure 26. The **rewards, transition probability, actions** and **discount factor** will be the similar to that in Part 1.

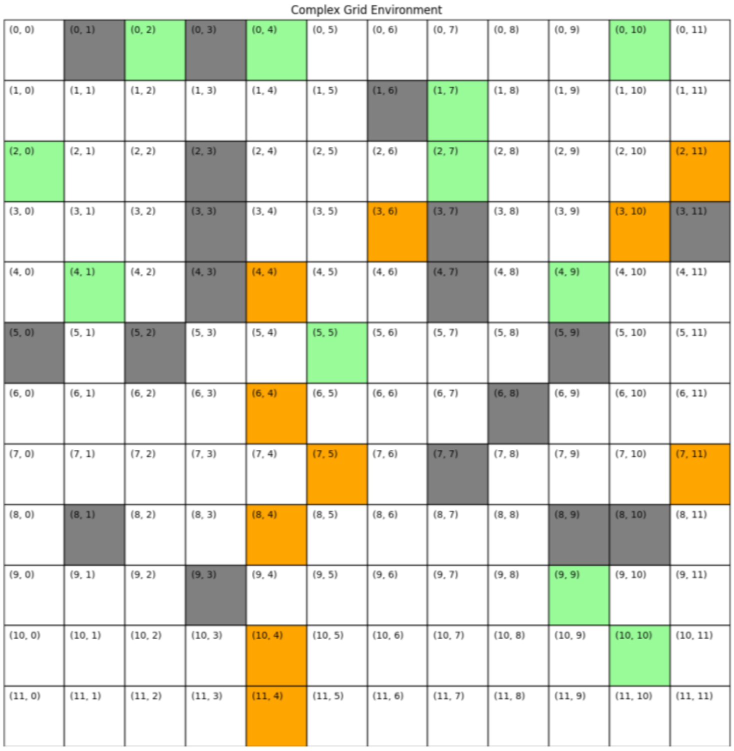


Figure 26. 12x12 Complex Environment Grid

Value Iteration with different c values

Generating the plots is similar to that in Part 1, by using the same Value Iteration algorithm for different values of c. The trend is similar as in Part 1, utility values for non-wall positions converge more smoothly when the value of c is higher but require more iterations.

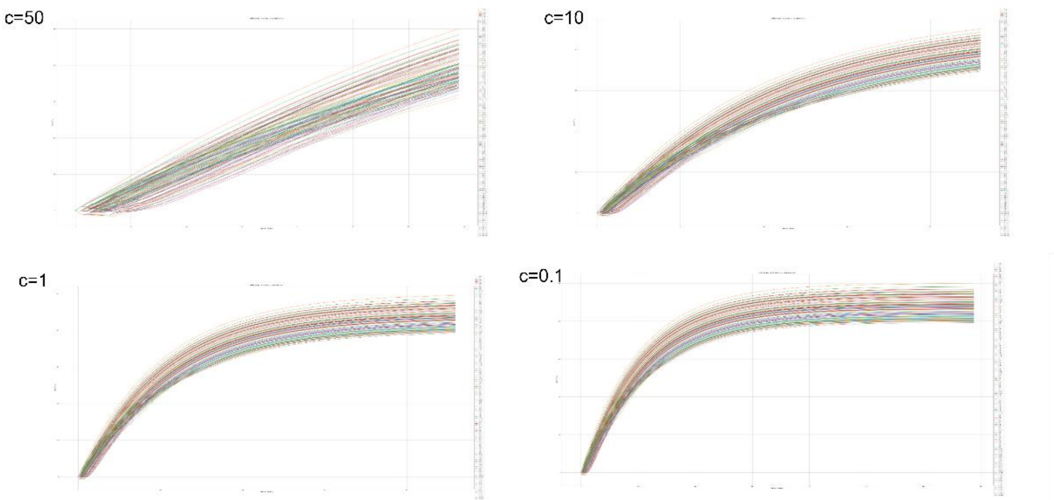


Figure 27. Plots of number of iterations against utility value for non-wall position in complex grid where c = [50,10,1,0.1]

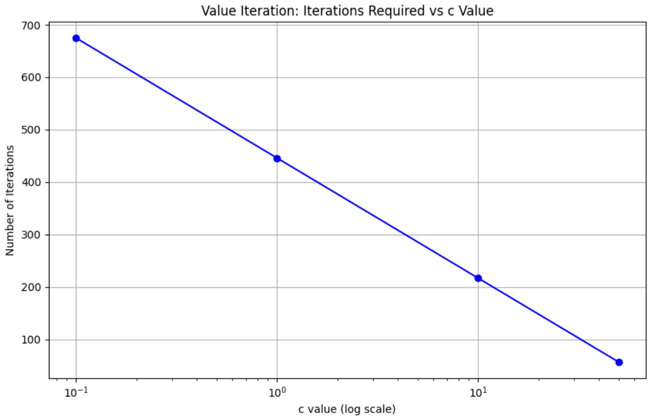


Figure 28. Plot of c value against number of iterations in complex grid

We note that for both complex and simple grid, it required the same number of iterations to reach convergence. As seen in Figure *9*, it requires the same number of iterations to converge for a simple 6x6 grid as it takes for a complex 12x12 grid in Figure *29*. This suggests that the convergence rate might be more strongly influenced by the discount factor γ and the threshold parameter than by the size of the state space.

```
Experiment 1: Value Iteration with different values of c
--------------------------------------------------------
Running Value Iteration with c=50 (threshold=0.5)...
Value Iteration converged after 70 iterations
Running Value Iteration with c=10 (threshold=0.1)...
Value Iteration converged after 231 iterations
Running Value Iteration with c=1 (threshold=0.01)...
Value Iteration converged after 460 iterations
Running Value Iteration with c=0.1 (threshold=0.001)...
Value Iteration converged after 689 iterations
```

Figure 29. Terminal output from Experiment 1

Results of Value Iteration with c=1 (threshold=0.01)

Based on the experimental results, we note that using Value Iteration, with a bigger state size, it does not affect the number of iterations needed to converge. I selected c=1 as a good balance between computational efficiency and solution accuracy, giving a threshold of 0.01. With this threshold, Value Iteration converged after 460 iterations.

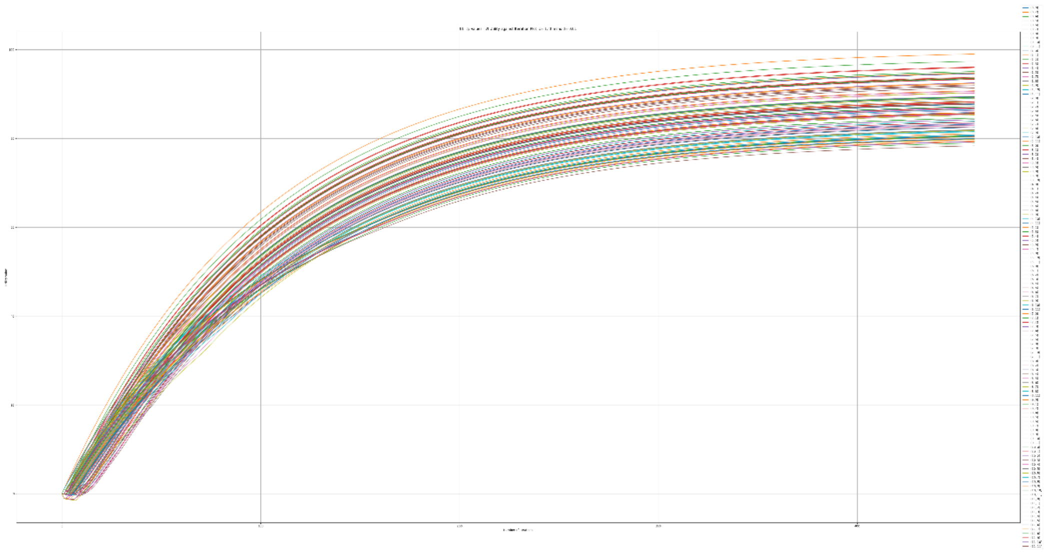


Figure 30. Value Iteration Plot of utility estimates as a function of the number of iterations in complex grid, c =1

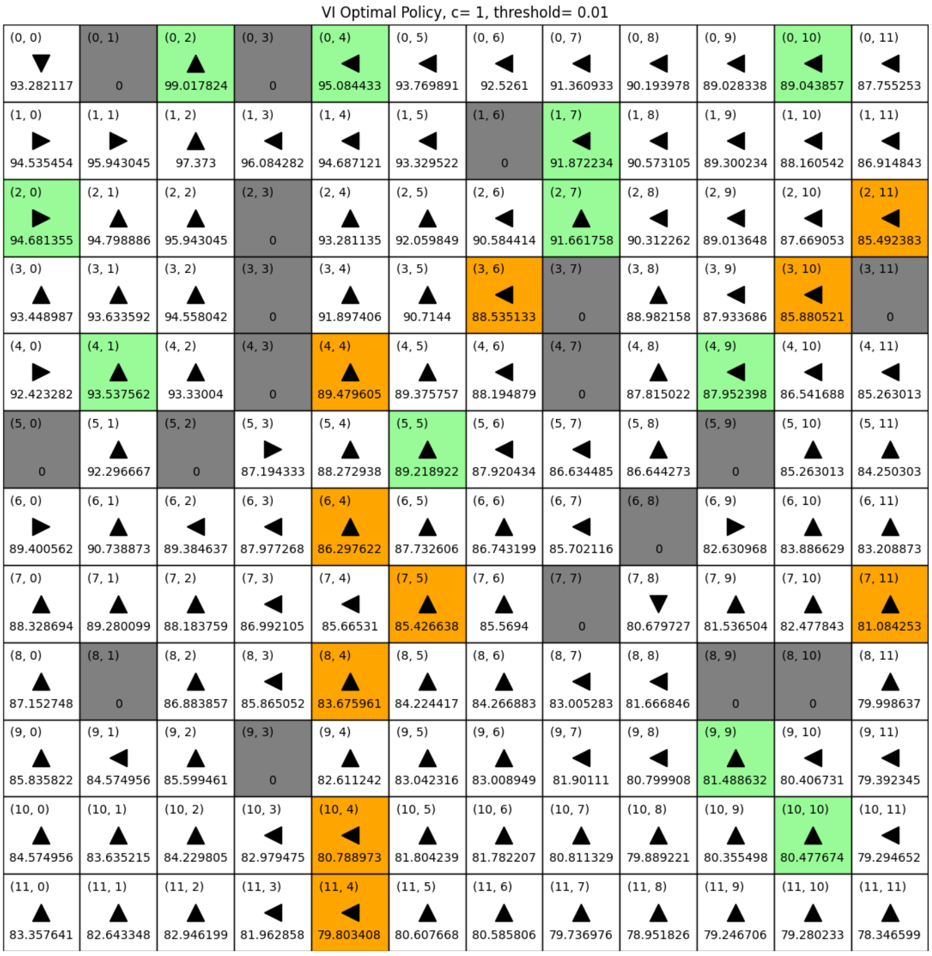


Figure 31. Plot of optimal policy and Utilities of all states in complex grid, c = 1

UTILITY VALUE OF ALL STATES

	0	1	2	3	4	5	6	7	8	9	10	11
0	93.282117	0	99.017824	0	95.084433	93.769891	92.526100	91.360933	90.193978	89.028338	89.043857	87.755253
1	94.535454	95.943045	97.373000	96.084282	94.687121	93.329522	0	91.872234	90.573105	89.300234	88.160542	86.914843
2	94.681355	94.798886	95.943045	0	93.281135	92.059849	90.584414	91.661758	90.312262	89.013648	87.669053	85.492383
3	93.448987	93.633592	94.558042	0	91.897406	90.714400	88.535133	0	88.982158	87.933686	85.880521	0
4	92.423282	93.537562	93.330040	0	89.479605	89.375757	88.194879	0	87.815022	87.952398	86.541688	85.263013
5	0	92.296667	0	87.194333	88.272938	89.218922	87.920434	86.634485	86.644273	0	85.263013	84.250303
6	89.400562	90.738873	89.384637	87.977268	86.297622	87.732606	86.743199	85.702116	0	82.630968	83.886629	83.208873
7	88.328694	89.280099	88.183759	86.992105	85.665310	85.426638	85.569400	0	80.679727	81.536504	82.477843	81.084253
8	87.152748	0	86.883857	85.865052	83.675961	84.224417	84.266883	83.005283	81.666846	0	0	79.998637
9	85.835822	84.574956	85.599461	0	82.611242	83.042316	83.008949	81.901110	80.799908	81.488632	80.406731	79.392345
10	84.574956	83.635215	84.229805	82.979475	80.788973	81.804239	81.782207	80.811329	79.889221	80.355498	80.477674	79.294652
11	83.357641	82.643348	82.946199	81.962858	79.803408	80.607668	80.585806	79.736976	78.951826	79.246706	79.280233	78.346599

Figure 32. Utilities of all States in markdown table form in complex grid, c = 1

Policy Iteration, Modified Policy with different k values

As for policy iteration, we implement a modified policy iteration which is more efficient than the regular policy iteration. This is seen in Figure 33, of the reference book in page 657. Where we use k iterations to produce the next utility estimate instead of doing exact policy evaluations.

The important point is that these equations are *linear*, because the "max" operator has been removed. For n states, we have n linear equations with n unknowns, which can be solved exactly in time $O(n^3)$ by standard linear algebra methods.

For small state spaces, policy evaluation using exact solution methods is often the most efficient approach. For large state spaces, $O(n^3)$ time might be prohibitive. Fortunately, it is not necessary to do *exact* policy evaluation. Instead, we can perform some number of simplified value iteration steps (simplified because the policy is fixed) to give a reasonably good approximation of the utilities. The simplified Bellman update for this process is

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

and this is repeated k times to produce the next utility estimate. The resulting algorithm is called **modified policy iteration**. It is often much more efficient than standard policy iteration or value iteration.

Figure 33. Reference book page 657

In Figure 34, the modified policy evaluation at line 165 uses k iterations instead of the previous condition in Figure 18, which only stops when delta < threshold in line 59.

```
138 def policy_evaluation_modified(policy, utilities, grid, rewards, actions, transition_probs, gamma, policy_eval_records, policy_improv_records, k_iterations):
139     """...
159     height, width = len(grid), len(grid[0])
160     # Record policy improvement utilities for tracked positions
161     for pos in policy_improv_records:
162         if 0 <= pos[0] < height and 0 <= pos[1] < width:
163             policy_improv_records[pos].append(utilities[pos[0]][pos[1]])
164     eval_iterations = 0
165     for _ in range(k_iterations):
166         eval_iterations += 1
167         new_utilities = np.zeros((height, width))
168         # Record policy evaluation utilities for tracked positions
169         for pos in policy_eval_records:
170             if 0 <= pos[0] < height and 0 <= pos[1] < width:
171                 policy_eval_records[pos].append(utilities[pos[0]][pos[1]])
172         for row in range(height):
173             for col in range(width):
174                 if grid[row][col] == 'WALL':
175                     continue
176                 # Default to UP if no policy defined
177                 action = policy.get((row, col), 'UP')
178                 # Calculate expected utility
179                 expected_utility = 0
180                 # For intended action
181                 new_row, new_col = move(row, col, action, grid)
182                 expected_utility += transition_probs['intended'] * utilities[new_row][new_col]
183                 # For side actions
184                 for side_action in side_actions(action):
185                     new_row, new_col = move(row, col, side_action, grid)
186                     expected_utility += transition_probs['side'] * utilities[new_row][new_col]
187                 new_utilities[row][col] = rewards[grid[row][col]] + gamma * expected_utility
188         utilities = new_utilities
189     return utilities, eval_iterations
```

Figure 34. Implementation of Modified Policy Evaluation.

The implementation of the Modified Policy Iteration is exactly similar to Figure 17, but it calls policy_evaluation_modified at line 103 instead of policy_evaluation. This is required as with bigger search spaces, the $O(n^3)$ search time is prohibitive. As with Part 1, we have used an arbitrary c value to demonstrate the large search time required for a bigger grid environment. In this case, we used c = 1, Figure 35 along with k values of [5, 10, 20, 50, 100]. In Figure 35, we also note that with a larger state size of a complex 12x12 grid, it takes more iterations for the Regular Policy Iteration algorithm to converge. For Regular Policy Iteration with c=1, took 1032 policy evaluation iterations to converge for a complex 12x12 grid in Figure 35 while it took 860 policy evaluation iterations to converge for a simple 6x6 grid in Figure 19. We also plotted the number of iterations against the k value and had the Regular Policy Iteration iterations represented as by a red line in Figure 36.

```
Experiment 2: Regular vs Modified Policy Iteration algorithm
-------------------------------------------------------
Selected c value: 1 with threshold: 0.01
Running Regular Policy Iteration with c = 1...
Policy Iteration converged after 1032 policy evaluation iterations and 7 policy improvement iterations

Varying k values for Modified Policy Iteration: [5, 10, 20, 50, 100]
Running Modified Policy Iteration with k = 5...
Modified Policy Iteration converged after 90 policy evaluation iterations and 18 policy improvement iterations
Running Modified Policy Iteration with k = 10...
Modified Policy Iteration converged after 120 policy evaluation iterations and 12 policy improvement iterations
Running Modified Policy Iteration with k = 20...
Modified Policy Iteration converged after 140 policy evaluation iterations and 7 policy improvement iterations
Running Modified Policy Iteration with k = 50...
Modified Policy Iteration converged after 350 policy evaluation iterations and 7 policy improvement iterations
Running Modified Policy Iteration with k = 100...
Modified Policy Iteration converged after 700 policy evaluation iterations and 7 policy improvement iterations
```

Figure 35. Terminal output for Experiment 2

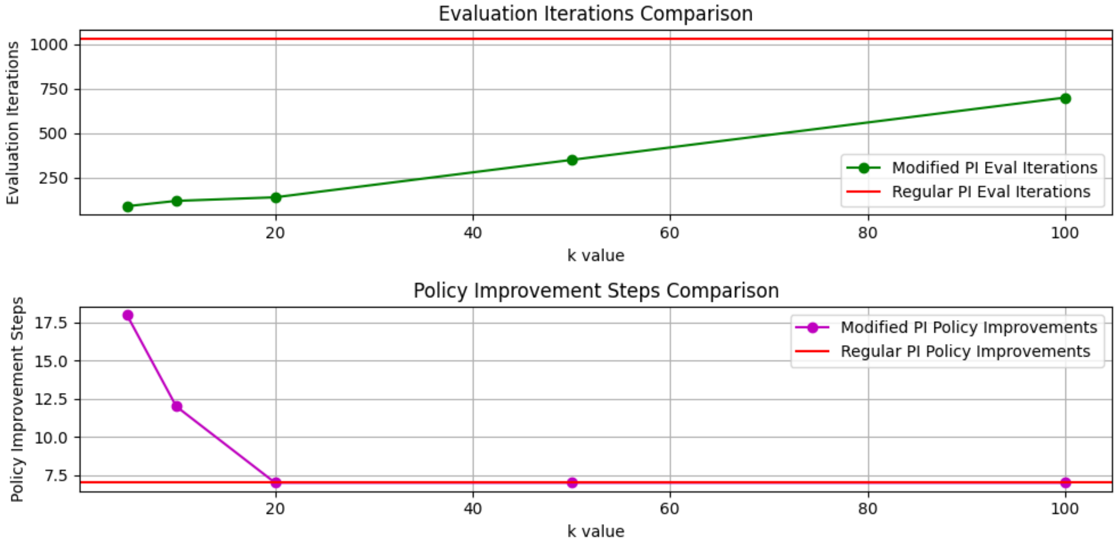


Figure 36. Regular and Modified Policy Evaluation (Eval and Improvement) Iterations against k value

As with Part 1, we will use the utility value against policy evaluation iterations to have a wider range of values to compare as compared to policy improvement iterations. In Figure 37, shows how different values of k affect the convergence of utility values in Modified Policy Iteration. As k increases, Modified Policy Iteration's convergence pattern begins to more closely resemble that of regular Policy Iteration. This is because with larger k values, the policy evaluation phase more thoroughly computes the utilities for the current policy before moving to the next policy improvement step. At k=20, we can see sharper transitions between policy improvements, indicating that the utility values haven't fully stabilized before the next policy change. As we increase to k=50 and k=100, the utility curves become smoother with more gradual transitions between policy improvements, approaching the behaviour of regular Policy Iteration which continues policy evaluation until convergence.

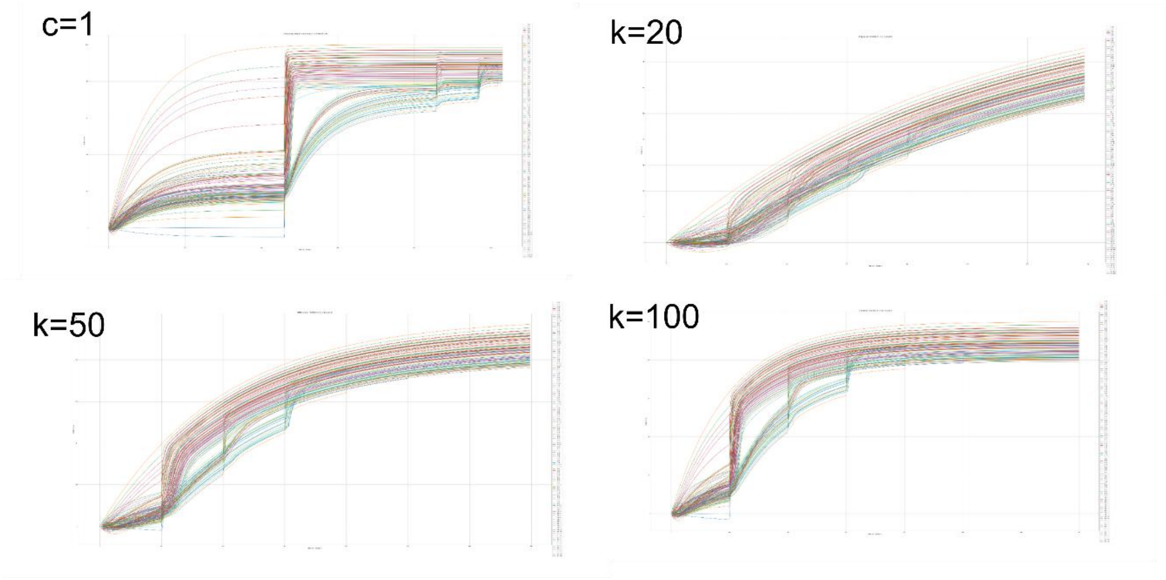


Figure 37. Plots of number of evaluation iterations against utility value for non-wall position where c = 1 and k = [20,50,100]

Results of Modified Policy Iteration with k=50

Based on the experimental results, we note that using the Regular Policy Iteration, with a bigger state space of a complex 12x12 grid, it takes more iterations to converge. Using the Modified Policy Iteration, I selected k=50 as a good balance between computational efficiency and solution accuracy. With this value, Modified Policy Iteration converged after 350 policy evaluation iterations and 7 policy improvement iterations.

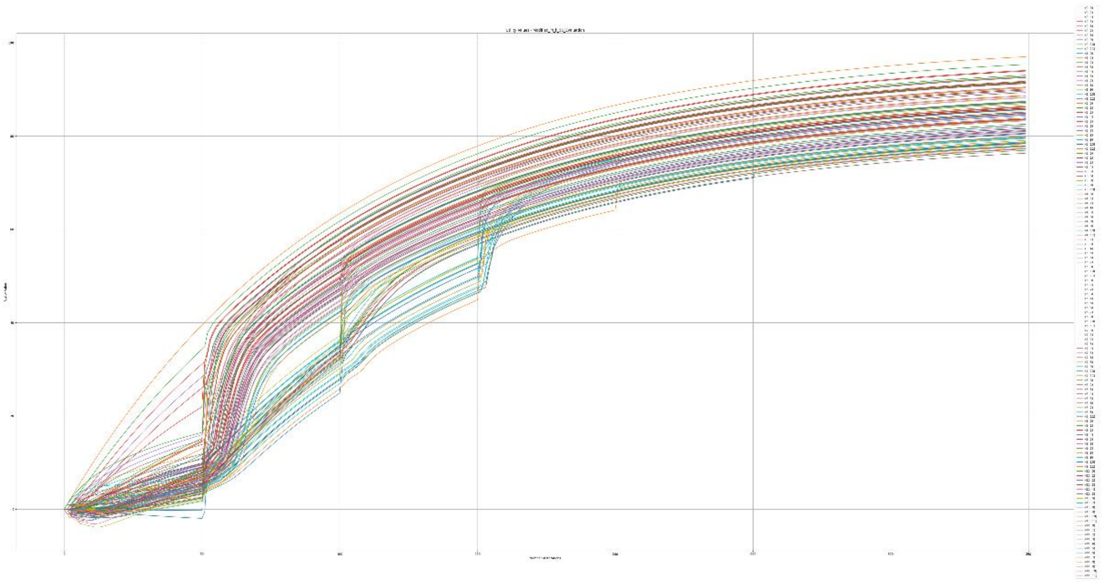


Figure 38. Modified Policy Iteration Plot of utility estimates as a function of the number of iterations (eval iterations), k=50

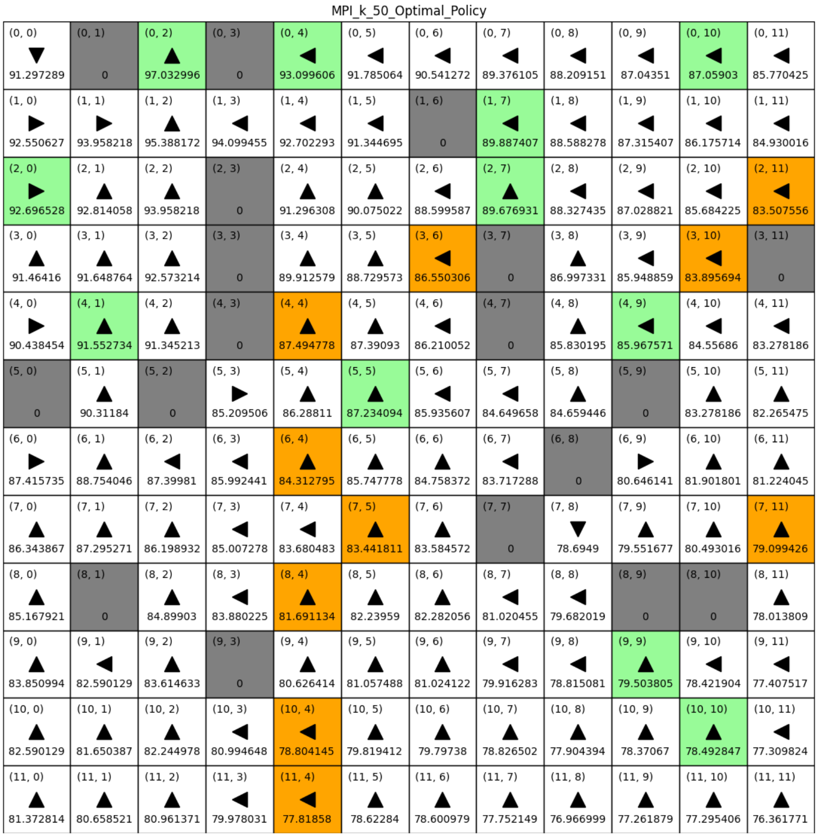


Figure 39. Plot of optimal policy and Utilities of all states, k=50

UTILITY VALUE OF ALL STATES

	0	1	2	3	4	5	6	7	8	9	10	11
0	91.297289	0	97.032996	0	93.099606	91.785064	90.541272	89.376105	88.209151	87.043510	87.059030	85.770425
1	92.550627	93.958218	95.388172	94.099455	92.702293	91.344695	0	89.887407	88.588278	87.315407	86.175714	84.930016
2	92.696528	92.814058	93.958218	0	91.296308	90.075022	88.599587	89.676931	88.327435	87.028821	85.684225	83.507556
3	91.464160	91.648764	92.573214	0	89.912579	88.729573	86.550306	0	86.997331	85.948859	83.895694	0
4	90.438454	91.552734	91.345213	0	87.494778	87.390930	86.210052	0	85.830195	85.967571	84.556860	83.278186
5	0	90.311840	0	85.209506	86.288110	87.234094	85.935607	84.649658	84.659446	0	83.278186	82.265475
6	87.415735	88.754046	87.399810	85.992441	84.312795	85.747778	84.758372	83.717288	0	80.646141	81.901801	81.224045
7	86.343867	87.295271	86.198932	85.007278	83.680483	83.441811	83.584572	0	78.694900	79.551677	80.493016	79.099426
8	85.167921	0	84.899030	83.880225	81.691134	82.239590	82.282056	81.020455	79.682019	0	0	78.013809
9	83.850994	82.590129	83.614633	0	80.626414	81.057488	81.024122	79.916283	78.815081	79.503805	78.421904	77.407517
10	82.590129	81.650387	82.244978	80.994648	78.804145	79.819412	79.797380	78.826502	77.904394	78.370670	78.492847	77.309824
11	81.372814	80.658521	80.961371	79.978031	77.818580	78.622840	78.600979	77.752149	76.966999	77.261879	77.295406	76.361771

Figure 40. Utilities of all States in markdown table form, k=50

Comparing Optimal Policies of VI(c=1) and Modified-PI(k=50)

As with Part 1, we also compared both policies derived from Value Iteration and Modified Policy Iteration. Both algorithms converged to identical policies for c=1 and k=50 respectively.

```
# Compare the policies
print("Comparing policies")
print("-----------------------------------------------------")
match_count = 0
total = 0
for row in range(len(complex_grid)):
    for col in range(len(complex_grid[0])):
        if complex_grid[row][col] != 'WALL':
            total += 1
            if vi_policy.get((row, col)) == mpi_policy.get((row, col)):
                match_count += 1
print(
    f"Policy agreement: {match_count}/{total} states ({match_count/total*100:.2f}%)\n\n")
```

```
Comparing policies
-----------------------------------------------------
Policy agreement: 126/126 states (100.00%)
```

Figure 41. Comparing optimal policies for Value Iteration c=1 and Modified Policy Iteration k=50 for complex 12x12 grid