NANYANG TECHNOLOGICAL UNIVERSITY

College of Computing and Data Science

SC4003 Intelligent Agents

Assignment 1

By Nema Aarushi (U2120814C)

Table of Contents

1. Overview of Solution Implementation

1.1 Project Objective

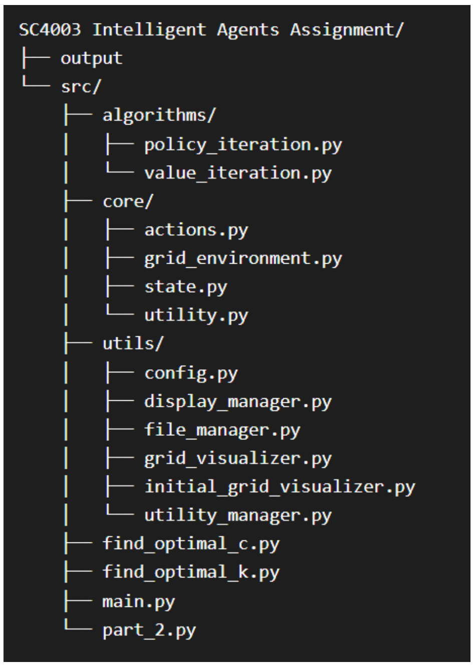
The main objective of this project is to implement and analyze two Markov Decision Policy methods: Value Iteration and Policy Iteration algorithms with the purpose of finding optimal policies for navigating a maze environment with probabilistic transition and rewards.

1.2 Summary of code structure and files

This assignment is implemented using Python. The entire assignment can be executed using the following command:

```
python -m src.main —-algorithm both --visualize
```

The figure below illustrates the file structure of the assignment.



```
SC4003 Intelligent Agents Assignment/
├── output
└── src/
    ├── algorithms/
    │   ├── policy_iteration.py
    │   └── value_iteration.py
    ├── core/
    │   ├── actions.py
    │   ├── grid_environment.py
    │   ├── state.py
    │   └── utility.py
    ├── utils/
    │   ├── config.py
    │   ├── display_manager.py
    │   ├── file_manager.py
    │   ├── grid_visualizer.py
    │   ├── initial_grid_visualizer.py
    │   └── utility_manager.py
    ├── find_optimal_c.py
    ├── find_optimal_k.py
    ├── main.py
    └── part_2.py
```

Figure 1. File structure of assignment

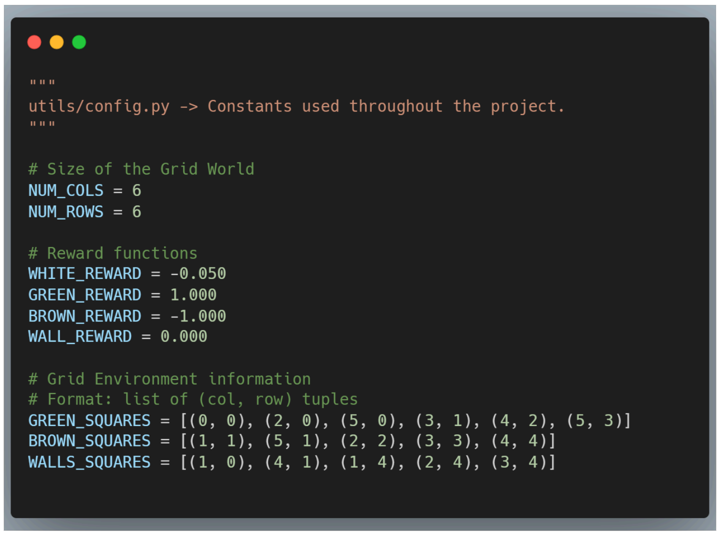
The implementation of this assignment is both modular and object oriented. There are three main packages:

1. `core` :
   Contains the core domain entities to represent the MDP environment
   a. `environment.py` : Represents the 6x6 grid word. It initializes the grid with the given reward values
   b. `state.py` : Represents a single cell in the environment (which is a grid).
   c. `action.py` : Represents four possible actions: UP, DOWN, LEFT, RIGHT. This class also contains a method to return a random action.
   d. `utility.py` : Pairs an action and its utility value. This is used to represent both the utility estimates and the policy for each state.

2. `algorithms` :
   Contains the two MDP solution algorithms
   a. `value_iteration.py` : Iteratively applies Bellman equation to update utility estimates until convergence. The convergence criterion is determined by the maximum change in utility being below a threshold calculated from the discount factor and epsilon value.
   b. `policy_iteration.py` : Iteratively evaluates and improves the policy until it converges to the optimal policy. The process involves alternating between policy evaluation, where state utilities are updated based on a fixed policy, and policy improvement, where the policy is updated by selecting actions that maximize expected utility.
3. `utils` :
   a. `constants.py` : Defines all constant values used throughout the project, including grid dimensions, reward values, transition probabilities, discount factor, and algorithm parameters.
   b. `utility_manager.py` : Provides utility calculation functions used by both algorithms, including methods to calculate the expected utility for each action, movement utilities considering the probabilistic transitions, and utilities for specific policy actions.
   c. `display_manager.py` : Handles all visualization and display functions, including rendering the grid environment, displaying the optimal policy, showing utility values, and presenting experiment configurations.
   d. `file_manager.py`: Manages file I/O operations, including saving utility estimates to CSV files and generating plots to visualize how utilities change across iterations

1.3 The Grid Environment

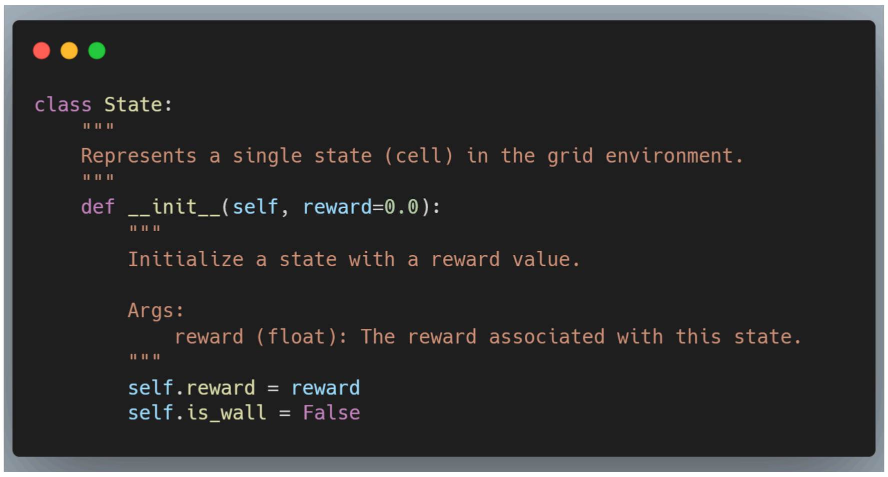
The MDP environment is built using the GridEnvironment class (in the utils/grid_environment.py file). It creates a 2-dimension grid world. Some variable values are defined in the utils/config.py file as shown in Figure 2.



```python
"""
utils/config.py -> Constants used throughout the project.
"""

# Size of the Grid World
NUM_COLS = 6
NUM_ROWS = 6

# Reward functions
WHITE_REWARD = -0.050
GREEN_REWARD = 1.000
BROWN_REWARD = -1.000
WALL_REWARD = 0.000

# Grid Environment information
# Format: list of (col, row) tuples
GREEN_SQUARES = [(0, 0), (2, 0), (5, 0), (3, 1), (4, 2), (5, 3)]
BROWN_SQUARES = [(1, 1), (5, 1), (2, 2), (3, 3), (4, 4)]
WALLS_SQUARES = [(1, 0), (4, 1), (1, 4), (2, 4), (3, 4)]
```

Figure 2. Code snippet of constants for the grid environment

The grid environment is represented as a 2-dimensional array of `State` objects. Each state has a reward value (-0.05) and a flag indicating whether it's a wall.

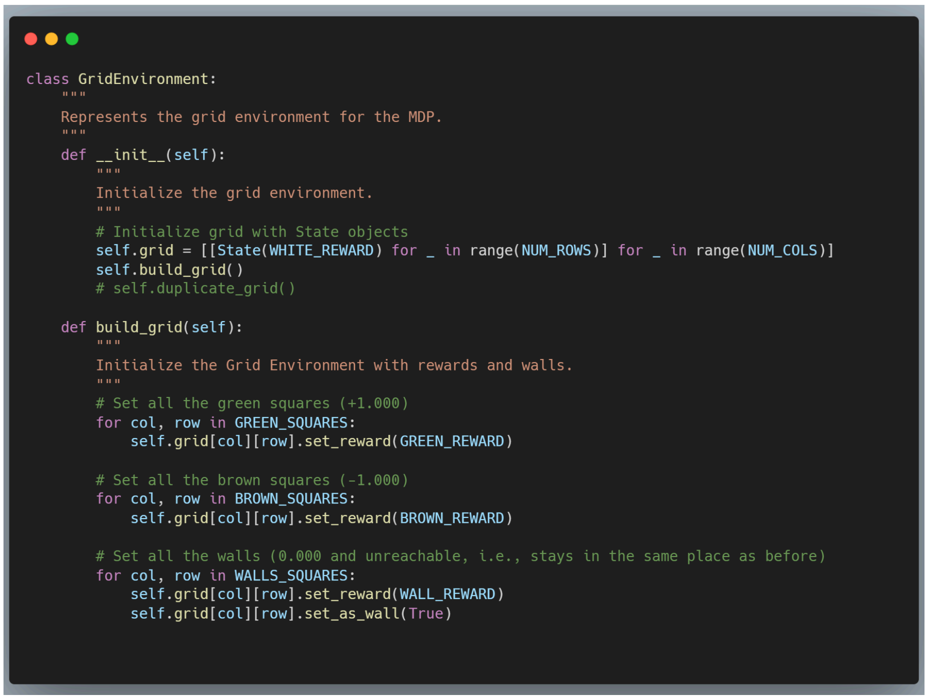


```python
class State:
    """
    Represents a single state (cell) in the grid environment.
    """
    def __init__(self, reward=0.0):
        """
        Initialize a state with a reward value.

        Args:
            reward (float): The reward associated with this state.
        """
        self.reward = reward
        self.is_wall = False
```

Figure 3. Code snippet of initialising a state / cell

When the grid is initialised, a uniform grid is created with all cells as white states. The environment is then customized using the `build_grid()` method based on the `config.py` file. Different state types are given the different rewards based on the configurations:

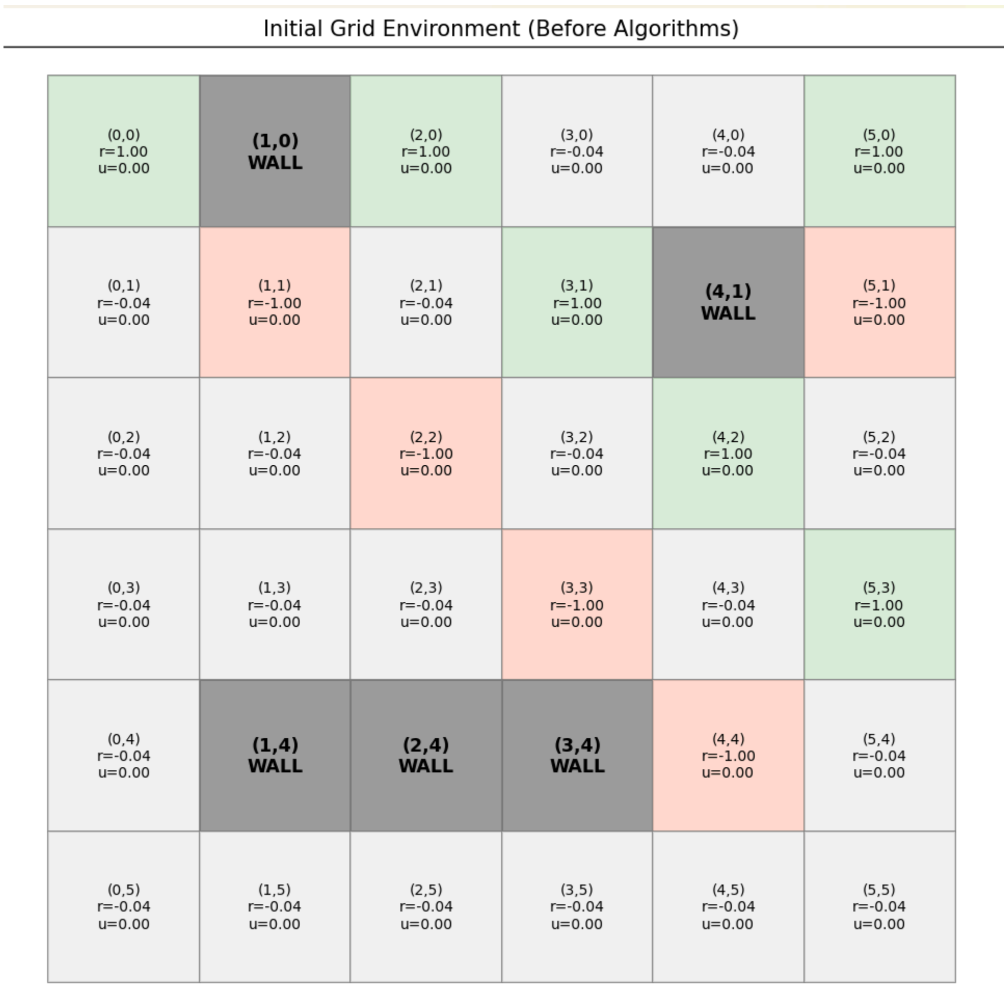
- Green states (reward +1.0) are placed at predefined locations for goal states
- Brown states (reward -1.0) are positioned to represent penalty areas
- Wall states (reward 0.0) are designated as impassable obstacles
- White states (reward -0.05) are already initialised



```python
class GridEnvironment:
    """
    Represents the grid environment for the MDP.
    """
    def __init__(self):
        """
        Initialize the grid environment.
        """
        # Initialize grid with State objects
        self.grid = [[State(WHITE_REWARD) for _ in range(NUM_ROWS)] for _ in range(NUM_COLS)]
        self.build_grid()
        # self.duplicate_grid()

    def build_grid(self):
        """
        Initialize the Grid Environment with rewards and walls.
        """
        # Set all the green squares (+1.000)
        for col, row in GREEN_SQUARES:
            self.grid[col][row].set_reward(GREEN_REWARD)

        # Set all the brown squares (-1.000)
        for col, row in BROWN_SQUARES:
            self.grid[col][row].set_reward(BROWN_REWARD)

        # Set all the walls (0.000 and unreachable, i.e., stays in the same place as before)
        for col, row in WALLS_SQUARES:
            self.grid[col][row].set_reward(WALL_REWARD)
            self.grid[col][row].set_as_wall(True)
```

Figure 4. Code snippet of initializing the grid environment

This finally results in the following initial grid environment:


Initial Grid Environment (Before Algorithms)

(0,0) r=1.00 u=0.00	**(1,0) WALL**	(2,0) r=1.00 u=0.00	(3,0) r=-0.04 u=0.00	(4,0) r=-0.04 u=0.00	(5,0) r=1.00 u=0.00
(0,1) r=-0.04 u=0.00	(1,1) r=-1.00 u=0.00	(2,1) r=-0.04 u=0.00	(3,1) r=1.00 u=0.00	**(4,1) WALL**	(5,1) r=-1.00 u=0.00
(0,2) r=-0.04 u=0.00	(1,2) r=-0.04 u=0.00	(2,2) r=-1.00 u=0.00	(3,2) r=-0.04 u=0.00	(4,2) r=1.00 u=0.00	(5,2) r=-0.04 u=0.00
(0,3) r=-0.04 u=0.00	(1,3) r=-0.04 u=0.00	(2,3) r=-0.04 u=0.00	(3,3) r=-1.00 u=0.00	(4,3) r=-0.04 u=0.00	(5,3) r=1.00 u=0.00
(0,4) r=-0.04 u=0.00	**(1,4) WALL**	**(2,4) WALL**	**(3,4) WALL**	(4,4) r=-1.00 u=0.00	(5,4) r=-0.04 u=0.00
(0,5) r=-0.04 u=0.00	(1,5) r=-0.04 u=0.00	(2,5) r=-0.04 u=0.00	(3,5) r=-0.04 u=0.00	(4,5) r=-0.04 u=0.00	(5,5) r=-0.04 u=0.00

Figure 5. Note that the tuple format is (col, row)

Here we observe that the initial rewards have been set as mentioned in the assignment guide and the utilities are assigned to 0.

2. Value Iteration Method

Value Iteration is an algorithm used to find the optimal policy in a Markov Decision Process (MDP) by iteratively computing the utility values of states where we repeatedly apply the Bellman update equation until convergence, at which point it can determine the optimal action for each state.

The utility of taking action a in state s is calculated as:

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

Figure 6. The core principle behind value iteration is the Bellman equation

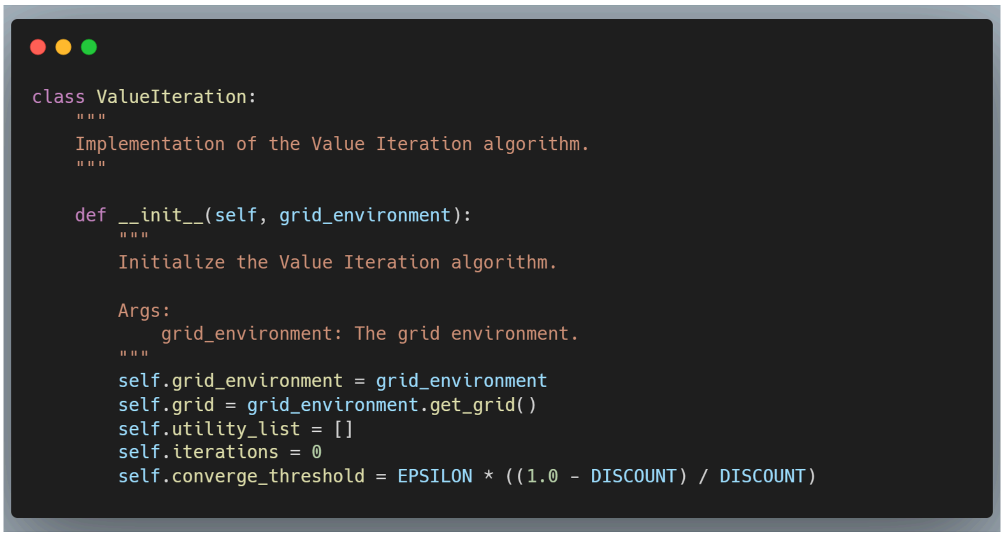
Where:

- `U(s)` -> utility of state `s`.
- `R(s)` -> immediate reward for being in state `s` -0.05 for white spaces, 1.0 for green spaces, -1.0 for brown spaces
- `γ` -> discount factor (0.99 in our implementation)
- `P(s'|s,a)` -> probability of transitioning to state `s` when taking action `a` in state `s` 0.8 to move in intended direction, 0.1 to move perpendicular.

2.1 Initialization of Environment

The algorithm begins by initializing the following components:

1. Two utility arrays:
   - currUtilArr -> Stores the current utility values of all states
   - newUtilArr -> Stores the update utility values after each iteration
2. All states are initialized with zero utility (as shown in Figure 5)
3. A convergence threshold is also initialised (Further discussed in section 2.3)



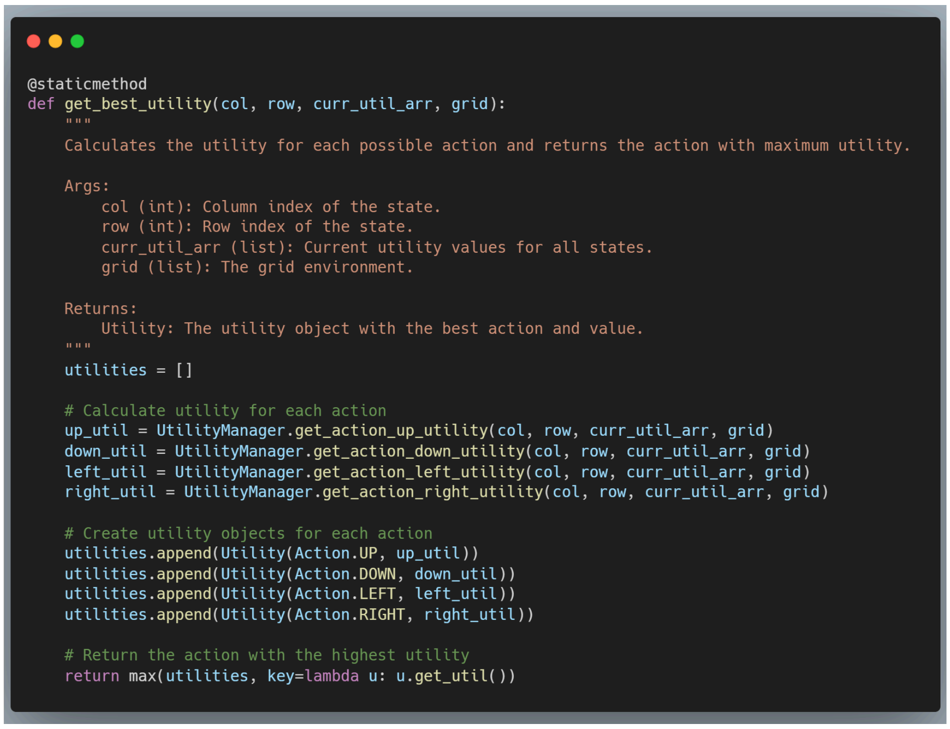
```python
class ValueIteration:
    """
    Implementation of the Value Iteration algorithm.
    """

    def __init__(self, grid_environment):
        """
        Initialize the Value Iteration algorithm.

        Args:
            grid_environment: The grid environment.
        """
        self.grid_environment = grid_environment
        self.grid = grid_environment.get_grid()
        self.utility_list = []
        self.iterations = 0
        self.converge_threshold = EPSILON * ((1.0 - DISCOUNT) / DISCOUNT)
```

Figure 7. Code snippet of Value Iteration Initialization

2.2 Bellman Update Implementation

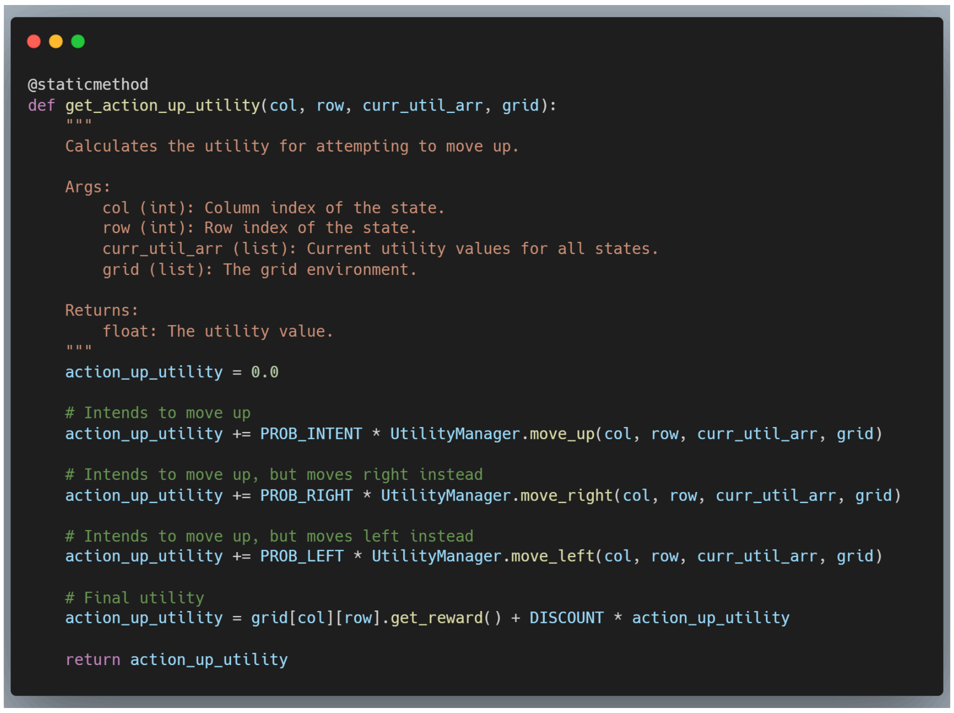
The core of the Value Iteration Algorithm is the Bellman Update, which is implemented as: For each state in the grid environment (excluding walls):

1. Calculate the utility for each possible actions (UP, DOWN, RIGHT, LEFT)
2. Select the action that yields the maximum expected utility
3. Update the state's utility value with the reward plus the discounted expected utility



```python
@staticmethod
def get_best_utility(col, row, curr_util_arr, grid):
    """
    Calculates the utility for each possible action and returns the action with maximum utility.

    Args:
        col (int): Column index of the state.
        row (int): Row index of the state.
        curr_util_arr (list): Current utility values for all states.
        grid (list): The grid environment.

    Returns:
        Utility: The utility object with the best action and value.
    """
    utilities = []

    # Calculate utility for each action
    up_util = UtilityManager.get_action_up_utility(col, row, curr_util_arr, grid)
    down_util = UtilityManager.get_action_down_utility(col, row, curr_util_arr, grid)
    left_util = UtilityManager.get_action_left_utility(col, row, curr_util_arr, grid)
    right_util = UtilityManager.get_action_right_utility(col, row, curr_util_arr, grid)

    # Create utility objects for each action
    utilities.append(Utility(Action.UP, up_util))
    utilities.append(Utility(Action.DOWN, down_util))
    utilities.append(Utility(Action.LEFT, left_util))
    utilities.append(Utility(Action.RIGHT, right_util))

    # Return the action with the highest utility
    return max(utilities, key=lambda u: u.get_util())
```

Figure 8. Code snippet of calculating the utility for taking each actions and finding the best action

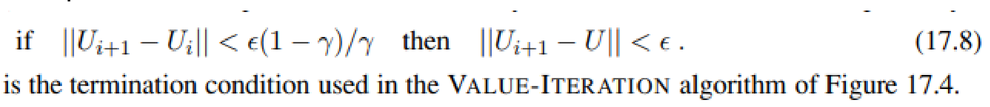


```python
@staticmethod
def get_action_up_utility(col, row, curr_util_arr, grid):
    """
    Calculates the utility for attempting to move up.

    Args:
        col (int): Column index of the state.
        row (int): Row index of the state.
        curr_util_arr (list): Current utility values for all states.
        grid (list): The grid environment.

    Returns:
        float: The utility value.
    """
    action_up_utility = 0.0

    # Intends to move up
    action_up_utility += PROB_INTENT * UtilityManager.move_up(col, row, curr_util_arr, grid)

    # Intends to move up, but moves right instead
    action_up_utility += PROB_RIGHT * UtilityManager.move_right(col, row, curr_util_arr, grid)

    # Intends to move up, but moves left instead
    action_up_utility += PROB_LEFT * UtilityManager.move_left(col, row, curr_util_arr, grid)

    # Final utility
    action_up_utility = grid[col][row].get_reward() + DISCOUNT * action_up_utility

    return action_up_utility
```

Figure 9. Code snippet of calculating the utility for taking action UP

2.3 Convergence Criteria

After each Iteration, the algorithm calculates delta, which is the maximum change in utility value across all states. The algorithm is looped over until we hit the convergence threshold. This threshold is derived from the theoretical analysis presented in the textbook "Artificial Intelligence: A Modern Approach" by Russell and Norvig (3rd edition, 2010). Specifically referring to Chapter 17 Section 2.3:



$$\text{if} \quad ||U_{i+1} - U_i|| < \epsilon(1-\gamma)/\gamma \quad \text{then} \quad ||U_{i+1} - U|| < \epsilon \,. \qquad (17.8)$$

This is the termination condition used in the VALUE-ITERATION algorithm of Figure 17.4.

Figure 10. Convergence criteria of value iteration as seen in the textbook

Where:

- ε is the maximum allowable error after k iterations
- γ is the discount factor (set to 0.99 in our project)

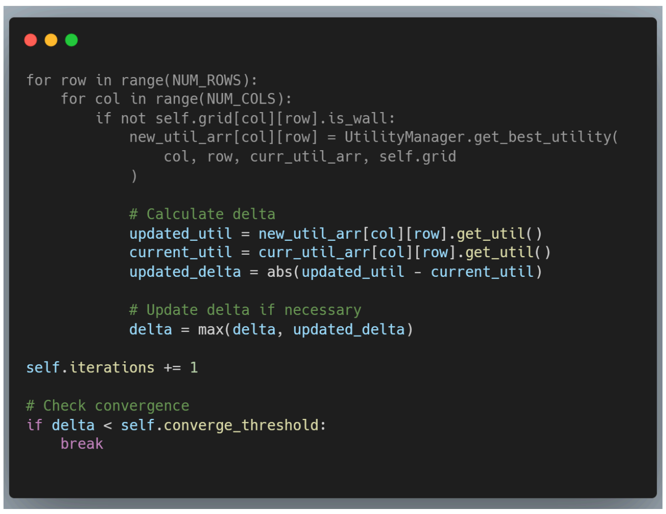
$$\varepsilon = C * R_{max}$$

Where

- C is a constant value, determined
- R_{max} is the maximum possible reward (in our case this is 1)

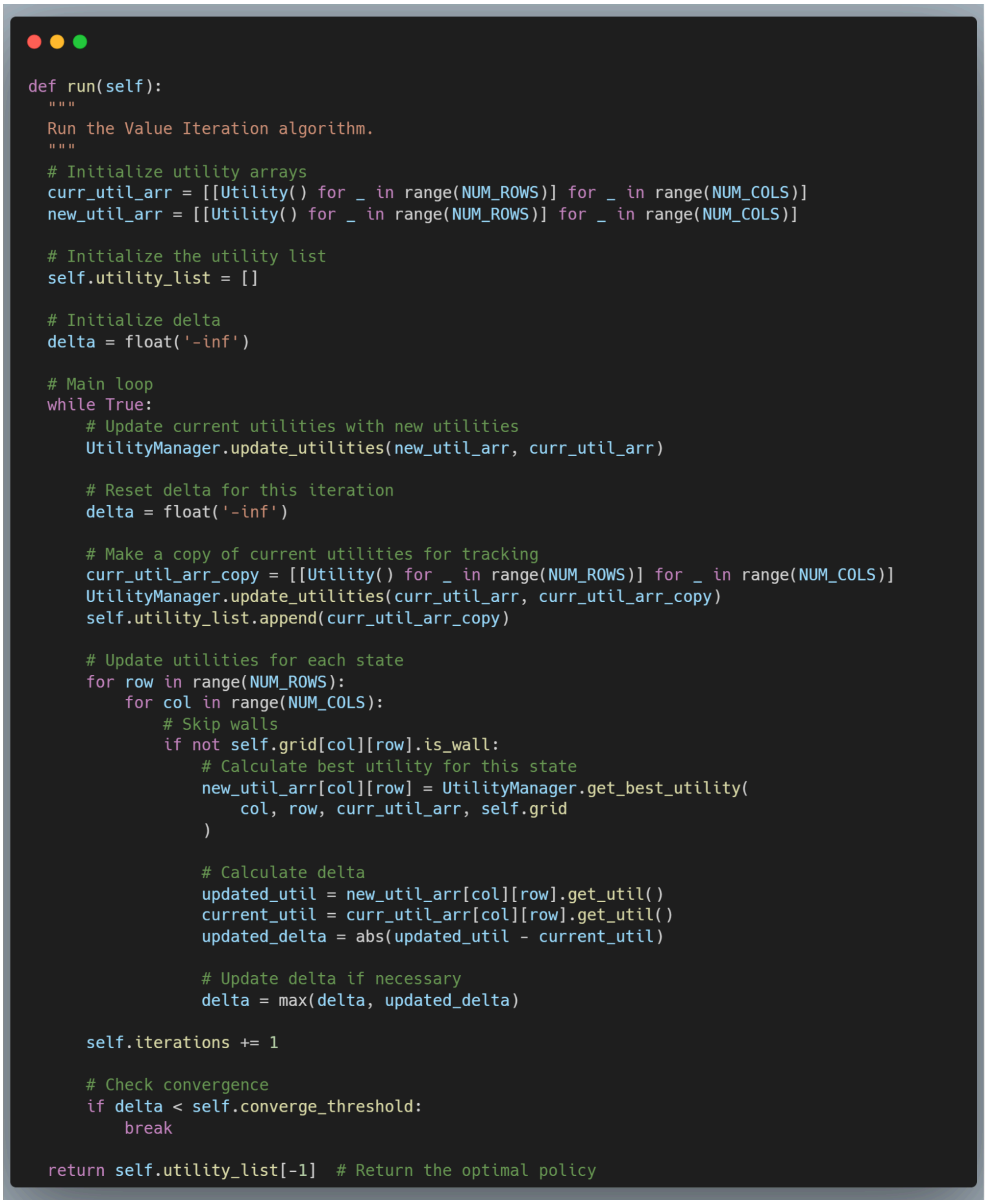
Since R_{max} is 1, we can say that $\varepsilon = C$

The formula in figure xx provides the stopping criteria that guarantees our solution is within ε of the optimal solution. By implementing this threshold, it is ensured that the value iteration algorithm terminates with utility estimates that support an optimal or near-optimal policy, with the error bounded by ε. The textbook also explains that as the discount factor approaches 1, the threshold becomes smaller, requiring more iterations to converge. In this project, experiments are done with multiple epsilon values to find the optimal convergence threshold (discussed in section 2.6). The selection of an appropriate c value is problem-dependent. For time-critical applications, a slightly larger value may be preferable, while applications requiring high precision should use a smaller value. Understanding this trade-off is essential for efficiently implementing Value Iteration in practical applications.



```
for row in range(NUM_ROWS):
    for col in range(NUM_COLS):
        if not self.grid[col][row].is_wall:
            new_util_arr[col][row] = UtilityManager.get_best_utility(
                col, row, curr_util_arr, self.grid
            )

            # Calculate delta
            updated_util = new_util_arr[col][row].get_util()
            current_util = curr_util_arr[col][row].get_util()
            updated_delta = abs(updated_util - current_util)

            # Update delta if necessary
            delta = max(delta, updated_delta)

self.iterations += 1

# Check convergence
if delta < self.converge_threshold:
    break
```

Figure 11. Code snippet of calculation of convergence criteria

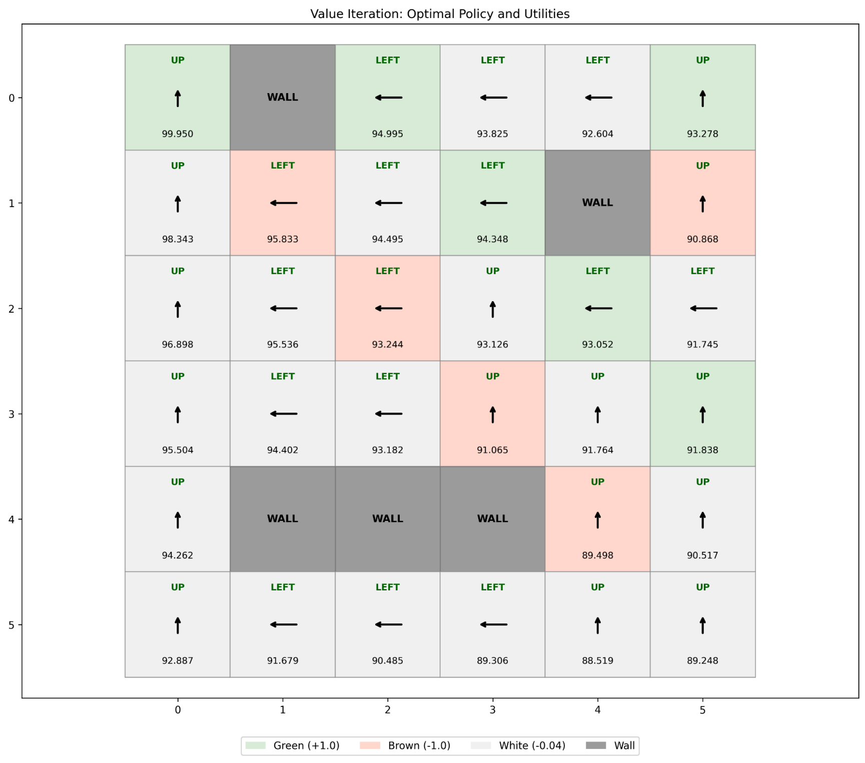


```python
def run(self):
    """
    Run the Value Iteration algorithm.
    """
    # Initialize utility arrays
    curr_util_arr = [[Utility() for _ in range(NUM_ROWS)] for _ in range(NUM_COLS)]
    new_util_arr = [[Utility() for _ in range(NUM_ROWS)] for _ in range(NUM_COLS)]

    # Initialize the utility list
    self.utility_list = []

    # Initialize delta
    delta = float('-inf')

    # Main loop
    while True:
        # Update current utilities with new utilities
        UtilityManager.update_utilities(new_util_arr, curr_util_arr)

        # Reset delta for this iteration
        delta = float('-inf')

        # Make a copy of current utilities for tracking
        curr_util_arr_copy = [[Utility() for _ in range(NUM_ROWS)] for _ in range(NUM_COLS)]
        UtilityManager.update_utilities(curr_util_arr, curr_util_arr_copy)
        self.utility_list.append(curr_util_arr_copy)

        # Update utilities for each state
        for row in range(NUM_ROWS):
            for col in range(NUM_COLS):
                # Skip walls
                if not self.grid[col][row].is_wall:
                    # Calculate best utility for this state
                    new_util_arr[col][row] = UtilityManager.get_best_utility(
                        col, row, curr_util_arr, self.grid
                    )

                    # Calculate delta
                    updated_util = new_util_arr[col][row].get_util()
                    current_util = curr_util_arr[col][row].get_util()
                    updated_delta = abs(updated_util - current_util)

                    # Update delta if necessary
                    delta = max(delta, updated_delta)

        self.iterations += 1

        # Check convergence
        if delta < self.converge_threshold:
            break

    return self.utility_list[-1]  # Return the optimal policy
```

Figure 12. Code snippet of the Value Iteration Controller Function

2.4 Optimal Policy and Utilities of all States

The figure below shows us the optimal policy and the utilities for all the states for our environment when using the parameters γ = 0.99 and c=0.05. (Optimal C value is calculated in section 2.6). The number of iterations to reach convergence is 757.
<u>NOTE THAT THE TUPLES ARE IN [COL][ROW] FORMAT IN THE FIGURE TO THE RIGHT</u>



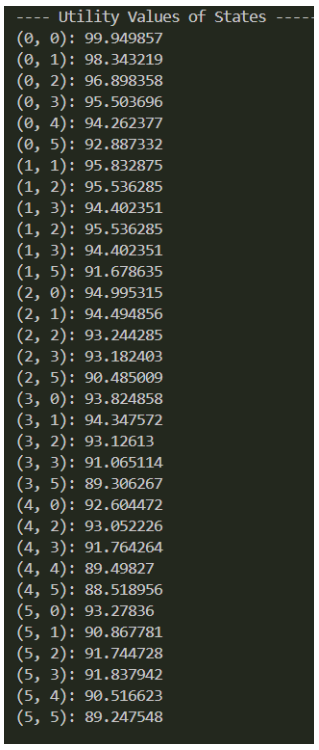

```
---- Utility Values of States ----
(0, 0): 99.949857
(0, 1): 98.343219
(0, 2): 96.898358
(0, 3): 95.503696
(0, 4): 94.262377
(0, 5): 92.887332
(1, 1): 95.832875
(1, 2): 95.536285
(1, 3): 94.402351
(1, 2): 95.536285
(1, 3): 94.402351
(1, 5): 91.678635
(2, 0): 94.995315
(2, 1): 94.494856
(2, 2): 93.244285
(2, 3): 93.182403
(2, 5): 90.485009
(3, 0): 93.824858
(3, 1): 94.347572
(3, 2): 93.12613
(3, 3): 91.065114
(3, 5): 89.306267
(4, 0): 92.604472
(4, 2): 93.052226
(4, 3): 91.764264
(4, 4): 89.49827
(4, 5): 88.518956
(5, 0): 93.27836
(5, 1): 90.867781
(5, 2): 91.744728
(5, 3): 91.837942
(5, 4): 90.516623
(5, 5): 89.247548
```

Figure 13. [Value Iteration] Optimal policy and utilities for all states (value iteration) at C = 0.05

2.5 Graphical Representation of Utility Estimates Over Iterations

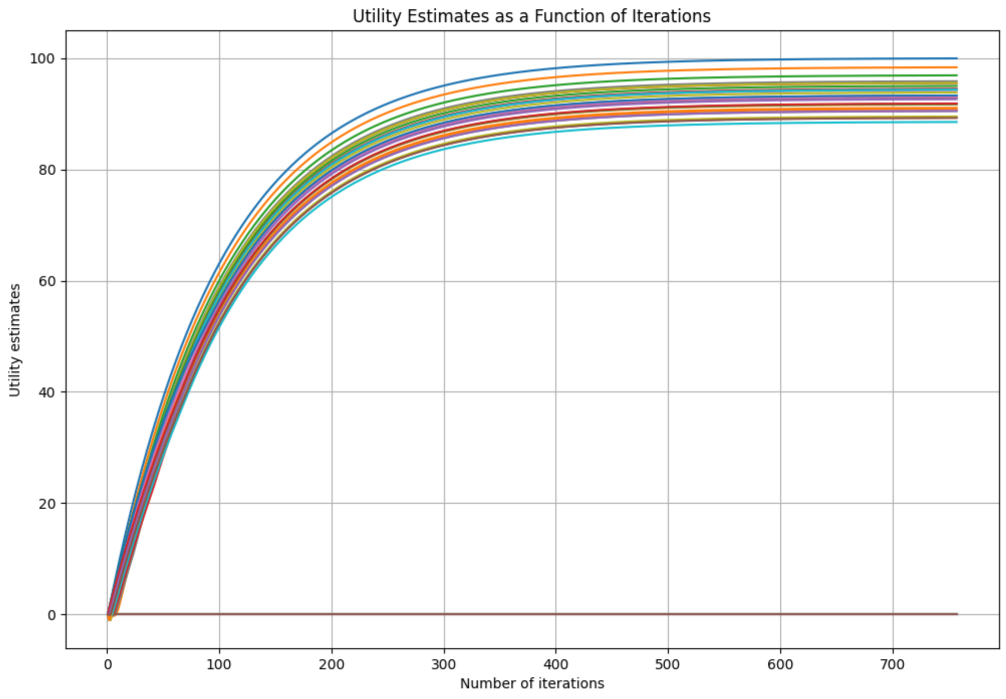
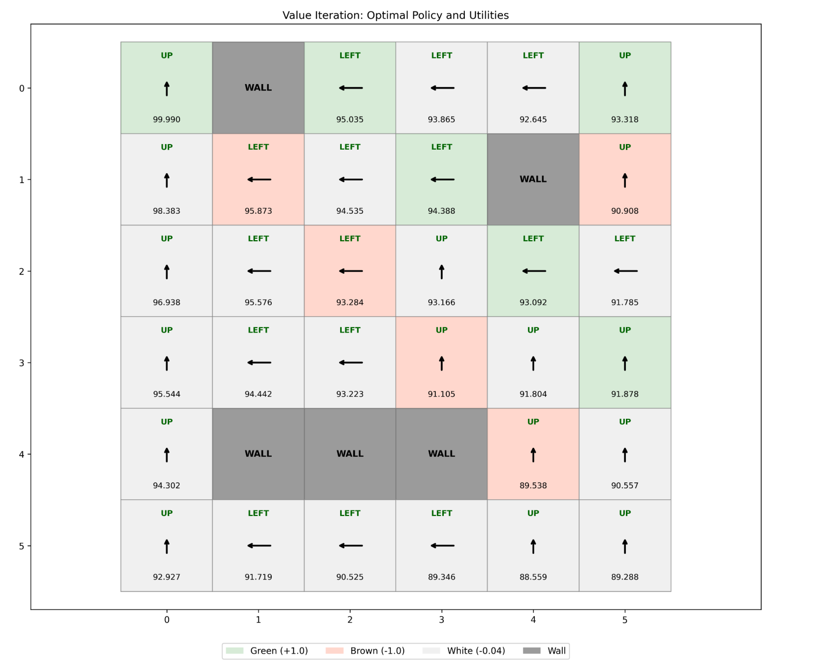


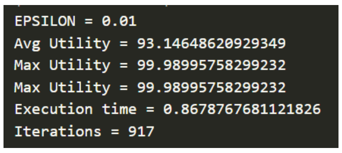
Figure 14. [Value Iteration] Utility estimates as a function of the number of iterations

2.6 Finding the Optimal value of C

To determine the optimal value of C for our grid-world MDP, an experiment is conducted to test a range of values from 0.01 to 60.0 (However, we have shown only a few in this report). For each value, the following metrics are measured using number of iterations required for convergence, average utility across all non-wall states, maximum utility value and execution time

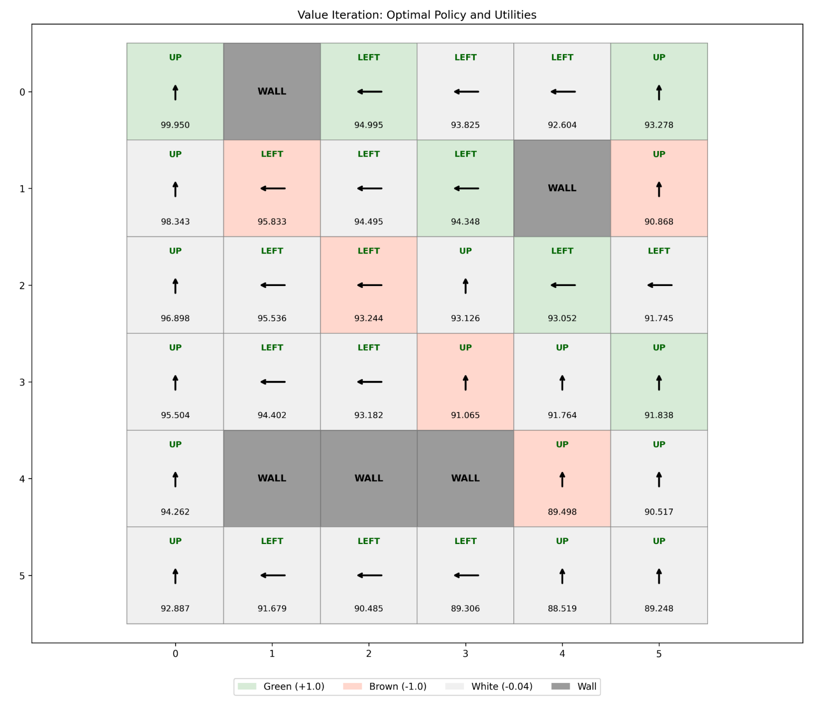
We first set C=0.01, so this implies that the epsilon value is also 0.01. For these settings, we observe that it takes 0.87 seconds to converge and 917 iterations to converge. Further details can be inferred from figures below

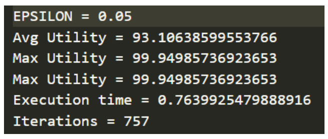

Value Iteration: Optimal Policy and Utilities
UP 99.990 | WALL | LEFT 95.035 | LEFT 93.865 | LEFT 92.645 | UP 93.318
UP 98.383 | LEFT 95.873 | LEFT 94.535 | LEFT 94.388 | WALL | UP 90.908
UP 96.938 | LEFT 95.576 | LEFT 93.284 | UP 93.166 | LEFT 93.092 | LEFT 91.785
UP 95.544 | LEFT 94.442 | LEFT 93.223 | UP 91.105 | UP 91.804 | UP 91.878
UP 94.302 | WALL | WALL | WALL | UP 89.538 | UP 90.557
UP 92.927 | LEFT 91.719 | LEFT 90.525 | LEFT 89.346 | UP 88.559 | UP 89.288
0
1
2
3
4
5
Green (+1.0)
Brown (-1.0)
White (-0.04)
Wall



```
EPSILON = 0.01
Avg Utility = 93.14648620929349
Max Utility = 99.98995758299232
Max Utility = 99.98995758299232
Execution time = 0.8678767681121826
Iterations = 917
```

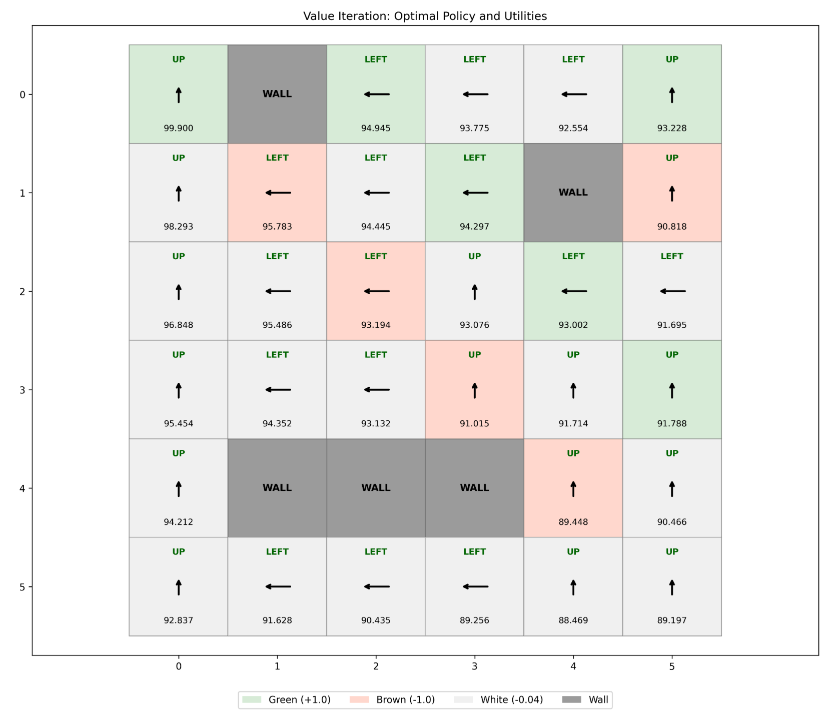
We first set C=0.05, so this implies that the epsilon value is also 0.05. For these settings, we observe that it takes 0.76 seconds to converge and 757 iterations to converge. Further details can be inferred from figures below

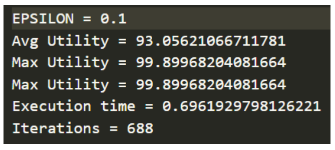

Value Iteration: Optimal Policy and Utilities
UP 99.950 | WALL | LEFT 94.995 | LEFT 93.825 | LEFT 92.604 | UP 93.278
UP 98.343 | LEFT 95.833 | LEFT 94.495 | LEFT 94.348 | WALL | UP 90.868
UP 96.898 | LEFT 95.536 | LEFT 93.244 | UP 93.126 | LEFT 93.052 | LEFT 91.745
UP 95.504 | LEFT 94.402 | LEFT 93.182 | UP 91.065 | UP 91.764 | UP 91.838
UP 94.262 | WALL | WALL | WALL | UP 89.498 | UP 90.517
UP 92.887 | LEFT 91.679 | LEFT 90.485 | LEFT 89.306 | UP 88.519 | UP 89.248
0
1
2
3
4
5
Green (+1.0)
Brown (-1.0)
White (-0.04)
Wall



```
EPSILON = 0.05
Avg Utility = 93.10638599553766
Max Utility = 99.94985736923653
Max Utility = 99.94985736923653
Execution time = 0.7639925479888916
Iterations = 757
```

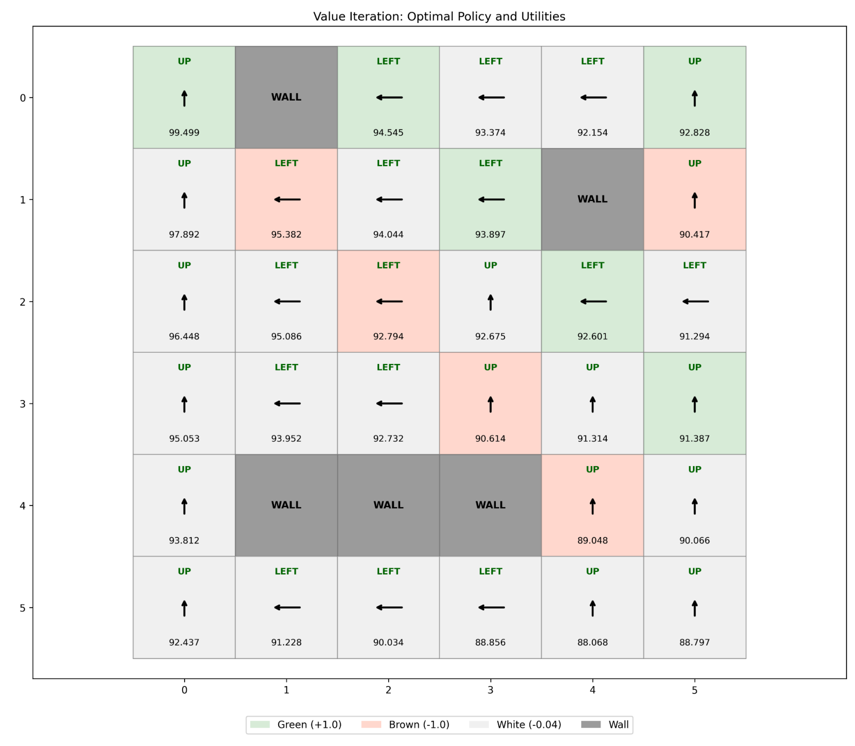
We first set C=0.1, so this implies that the epsilon value is also 0.1. For these settings, we observe that it takes 0.69 seconds to converge and 688 iterations to converge. Further details can be inferred from figures below

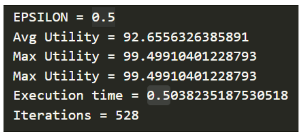




```
EPSILON = 0.1
Avg Utility = 93.05621066711781
Max Utility = 99.89968204081664
Max Utility = 99.89968204081664
Execution time = 0.6961929798126221
Iterations = 688
```

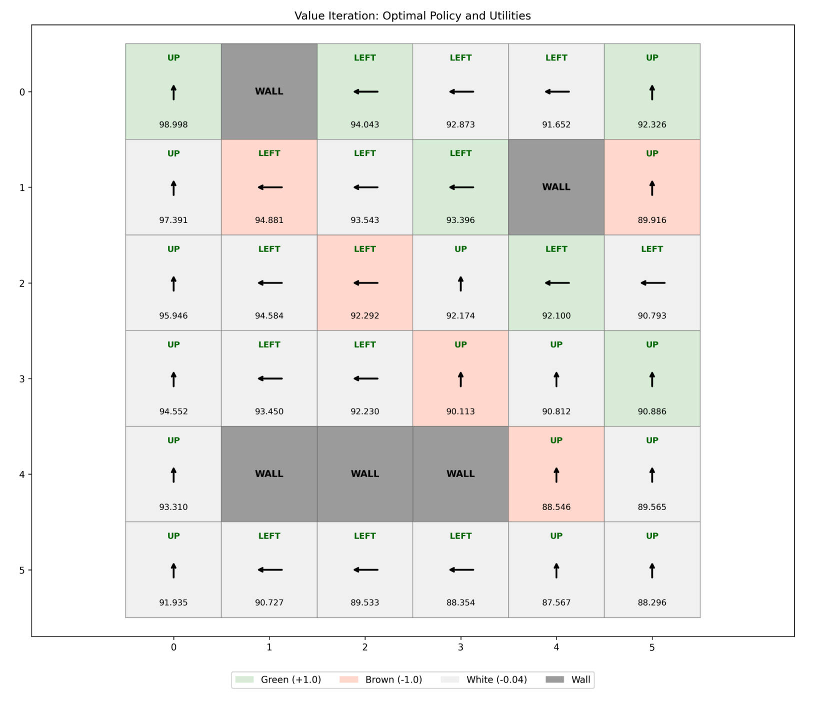
We first set C=0.5, so this implies that the epsilon value is also 0.5. For these settings, we observe that it takes 0.5 seconds to converge and 528 iterations to converge. Further details can be inferred from figures below

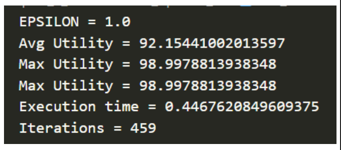




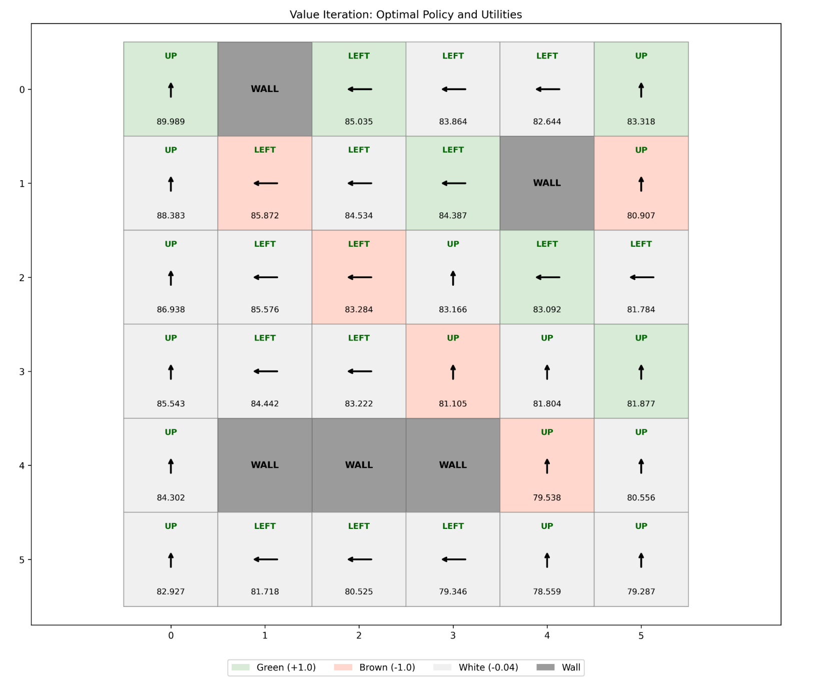
```
EPSILON = 0.5
Avg Utility = 92.6556326385891
Max Utility = 99.49910401228793
Max Utility = 99.49910401228793
Execution time = 0.5038235187530518
Iterations = 528
```

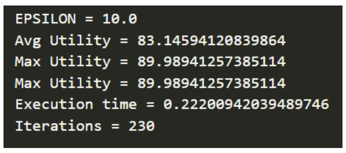
We first set C=1, so this implies that the epsilon value is also 1. For these settings, we observe that it takes 0.45 seconds to converge and 459 iterations to converge. Further details can be inferred from figures below




```
EPSILON = 1.0
Avg Utility = 92.15441002013597
Max Utility = 98.9978813938348
Max Utility = 98.9978813938348
Execution time = 0.4467620849609375
Iterations = 459
```

We first set C=10, so this implies that the epsilon value is also 0.01. For these settings, we observe that it takes 0.2 seconds to converge and 230 iterations to converge. Further details can be inferred from figures below




```
EPSILON = 10.0
Avg Utility = 83.14594120839864
Max Utility = 89.98941257385114
Max Utility = 89.98941257385114
Execution time = 0.22200942039489746
Iterations = 230
```

Conclusion

Based on the average utility metric, we should choose the optimal C value to be 0.01. However, we observe that for C=0.05, the number of iterations and the time taken to converge is lower than when C=0.01. Moreover, both max and avg utility are very close. Therefore, we choose the optimal C =0.05.

3. Policy Iteration Method

Policy Iteration is an algorithm for finding the optimal policy in a Markov Decision Process (MDP). Unlike Value Iteration, which works by directly estimating utilities, Policy Iteration alternates between policy evaluation (estimating utilities for a fixed policy) and policy improvement (updating the policy based on current utilities). This approach can converge faster than Value Iteration in certain scenarios.

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

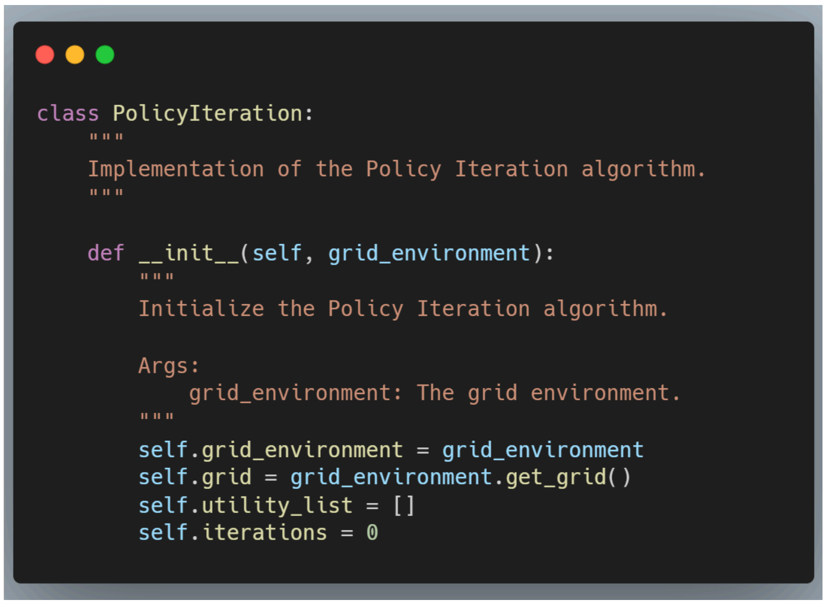
Figure 15. BellMan update for Policy Iteration

Where,

- U_{i+1} is the utility of state s
- $R(s)$ is the immediate reward for being in state s (-0.04 for white spaces, 1.0 for green spaces, -1.0 for brown spaces)
- γ is the discount factor (0.99 in our implementation)
- $P(s' \mid s, a)$ is the probability of transitioning to state s' when taking action a in state s (0.8 to move in the intended direction, 0.1 to move perpendicular)
- Π_i is the current policy which specifies an action for each state

3.1 Initialization of Environment

The Policy Iteration algorithm starts by initializing utility values and a random policy for each state:



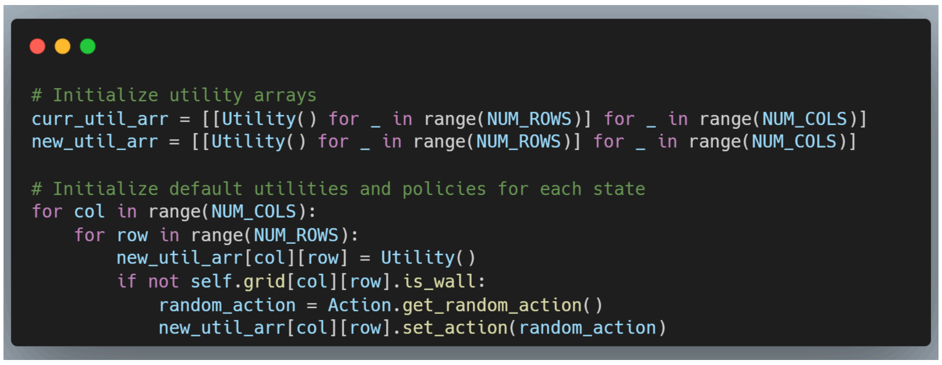
```python
class PolicyIteration:
    """
    Implementation of the Policy Iteration algorithm.
    """

    def __init__(self, grid_environment):
        """
        Initialize the Policy Iteration algorithm.

        Args:
            grid_environment: The grid environment.
        """
        self.grid_environment = grid_environment
        self.grid = grid_environment.get_grid()
        self.utility_list = []
        self.iterations = 0
```

Figure 16. Code snippet of Initialization of environment and # iterations for policy iteration

The utility array is initialized with zeros, and each non-wall state is assigned a random initial action:

- curr_util_arr -> Stores the current utility values of all states
- new_util_arr -> Stores the updated utility values after each iteration

All the states in the vector of the policy are instantiated with 0.0 as the utility values and random actions.

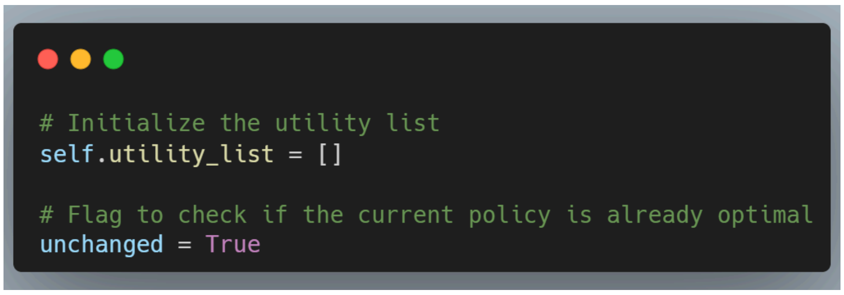


```python
# Initialize utility arrays
curr_util_arr = [[Utility() for _ in range(NUM_ROWS)] for _ in range(NUM_COLS)]
new_util_arr = [[Utility() for _ in range(NUM_ROWS)] for _ in range(NUM_COLS)]

# Initialize default utilities and policies for each state
for col in range(NUM_COLS):
    for row in range(NUM_ROWS):
        new_util_arr[col][row] = Utility()
        if not self.grid[col][row].is_wall:
            random_action = Action.get_random_action()
            new_util_arr[col][row].set_action(random_action)
```

Figure 17. Code snippet of Initializing utility arrays and assigning each state a random action

3.2 Convergence Criteria

A list named `utility_list` is initialized as a tracking mechanism to monitor the estimated value function across states during each algorithmic cycle. These values are later written to a csv file towards the end of each iteration. A terminator indicator variable, denoted as `unchanged`, serves as the convergence criterion, starting with a true value. This indicator switches to false during any iteration where the utility derived from the optimal policy action for any state exceeds that state's current utility value. This approach allows the algorithm to terminate when a complete iteration produces no utility improvements, signifying policy stability.



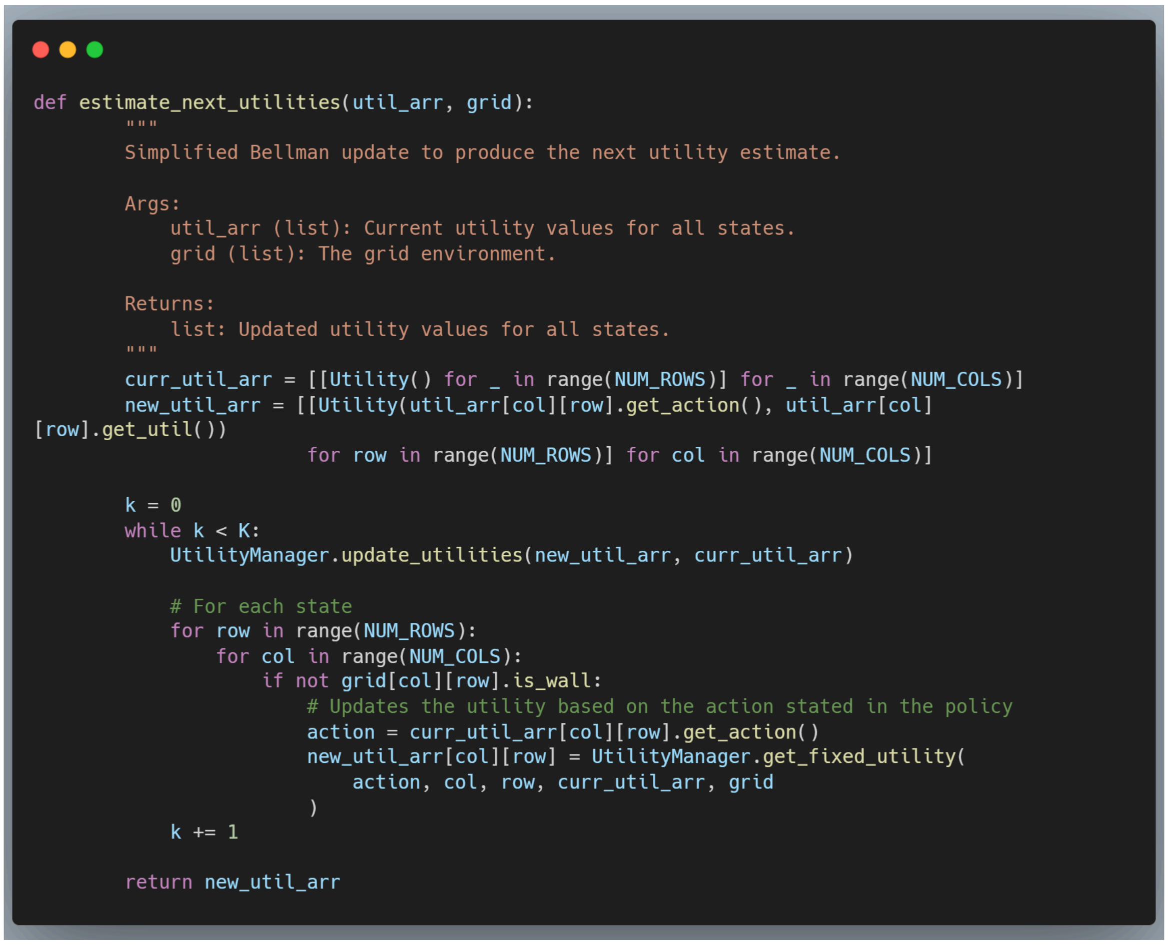
```python
# Initialize the utility list
self.utility_list = []

# Flag to check if the current policy is already optimal
unchanged = True
```

Figure 18. Code Snippet of Initializing utility arrays and assigning each state a random action

3.3 Policy Evaluation Phase

In the policy evaluation phase, we estimate the utility of each state under the current policy by executing a simplified Bellman update for a fixed number of iterations K. During each iteration:

- The utility manager updates all state utilities based on the current policy
- For each non-wall state in the grid:
    - Retrieves the action dictated by the current policy using `curr_util_arr[col][row].get_action()`
    - Calculates the expected utility of that action via `UtilityManager.compute_expected_utility()`
    - Updates the utility value in `new_util_arr` with this computed value
- The k counter is incremented after each complete iteration



```
def estimate_next_utilities(util_arr, grid):
        """
        Simplified Bellman update to produce the next utility estimate.

        Args:
            util_arr (list): Current utility values for all states.
            grid (list): The grid environment.

        Returns:
            list: Updated utility values for all states.
        """
        curr_util_arr = [[Utility() for _ in range(NUM_ROWS)] for _ in range(NUM_COLS)]
        new_util_arr = [[Utility(util_arr[col][row].get_action(), util_arr[col]
[row].get_util())
                        for row in range(NUM_ROWS)] for col in range(NUM_COLS)]

        k = 0
        while k < K:
            UtilityManager.update_utilities(new_util_arr, curr_util_arr)

            # For each state
            for row in range(NUM_ROWS):
                for col in range(NUM_COLS):
                    if not grid[col][row].is_wall:
                        # Updates the utility based on the action stated in the policy
                        action = curr_util_arr[col][row].get_action()
                        new_util_arr[col][row] = UtilityManager.get_fixed_utility(
                            action, col, row, curr_util_arr, grid
                        )
            k += 1

        return new_util_arr
```

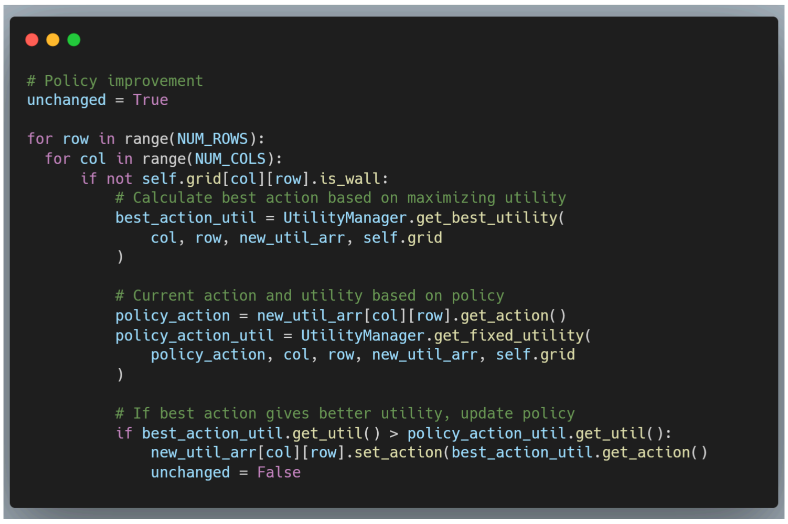
Figure 19. Code snippet of Policy evaluation phase

Unlike Value Iteration, we don't wait for convergence in this step - we simply perform a fixed number (K) of iterations. This significantly reduces the computational cost.

3.4 Policy Improvement Phase

In the policy improvement phase, we update the policy for each state by selecting the action that maximizes expected utility based on the current utility estimates. During this phase we iterate through each state in the grid environment. For each non-wall state:

- Calculates the optimal action by calling `UtilityManager.get_best_utility()`, which identifies the action that maximizes expected utility based on current utility values
- Retrieves the current policy action and its associated utility value
- Compares the utility of the best action against the utility of the current policy action
- If the best action yields higher utility:
    - Updates the policy by setting the state's action to the newly identified best action using `new_util_arr[col][row].set_action()`
    - Sets unchanged to False to indicate that the policy has been modified



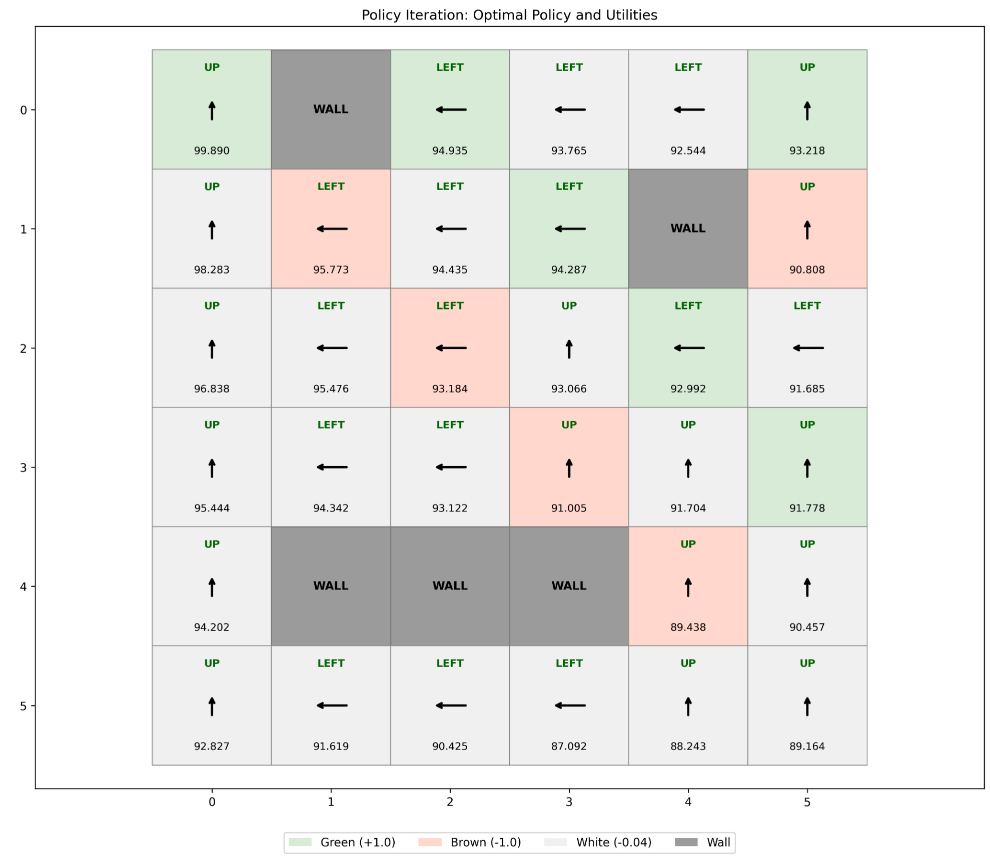
```python
# Policy improvement
unchanged = True

for row in range(NUM_ROWS):
  for col in range(NUM_COLS):
      if not self.grid[col][row].is_wall:
          # Calculate best action based on maximizing utility
          best_action_util = UtilityManager.get_best_utility(
              col, row, new_util_arr, self.grid
          )

          # Current action and utility based on policy
          policy_action = new_util_arr[col][row].get_action()
          policy_action_util = UtilityManager.get_fixed_utility(
              policy_action, col, row, new_util_arr, self.grid
          )

          # If best action gives better utility, update policy
          if best_action_util.get_util() > policy_action_util.get_util():
              new_util_arr[col][row].set_action(best_action_util.get_action())
              unchanged = False
```

Figure 20. Code snippet of policy improvement phase

We continue this process until the policy no longer changes, indicating that we've found the optimal policy.

3.5 Optimal Policy and Utilities of all States

The figure below shows the optimal policy and the utilities for all the states for our 6x6 grid environment when using the parameters γ = 0.99 and K = 100 (optimal K value is analysed in section 3.7). The number of iterations to convergence is approximately 6 to 8 (based on analysis). But in this run of policy iteration it took 9 iteration to converge.
<u>NOTE THAT THE TUPLES ARE IN [COL][ROW] FORMAT IN THE FIGURE TO THE RIGHT</u>



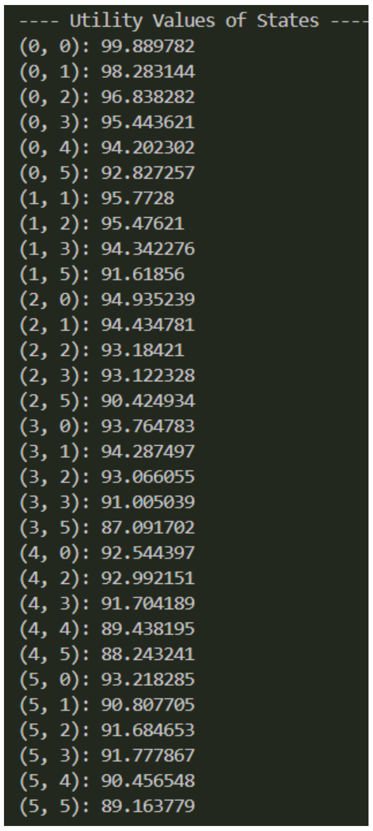

```
---- Utility Values of States ----
(0, 0): 99.889782
(0, 1): 98.283144
(0, 2): 96.838282
(0, 3): 95.443621
(0, 4): 94.202302
(0, 5): 92.827257
(1, 1): 95.7728
(1, 2): 95.47621
(1, 3): 94.342276
(1, 5): 91.61856
(2, 0): 94.935239
(2, 1): 94.434781
(2, 2): 93.18421
(2, 3): 93.122328
(2, 5): 90.424934
(3, 0): 93.764783
(3, 1): 94.287497
(3, 2): 93.066055
(3, 3): 91.005039
(3, 5): 87.091702
(4, 0): 92.544397
(4, 2): 92.992151
(4, 3): 91.704189
(4, 4): 89.438195
(4, 5): 88.243241
(5, 0): 93.218285
(5, 1): 90.807705
(5, 2): 91.684653
(5, 3): 91.777867
(5, 4): 90.456548
(5, 5): 89.163779
```

Figure 21. [Policy Iteration] The plot of optimal policy with utilities when “k” value is set to 100; total iterations required is approximately 5 to 8.

From the graph, we can observe:

- States closer to green squares (positive rewards) converge to higher utility values
- States closer to brown squares (negative rewards) converge to lower utility values

3.6 Graphical Representation of Utility Evolution Over Iterations

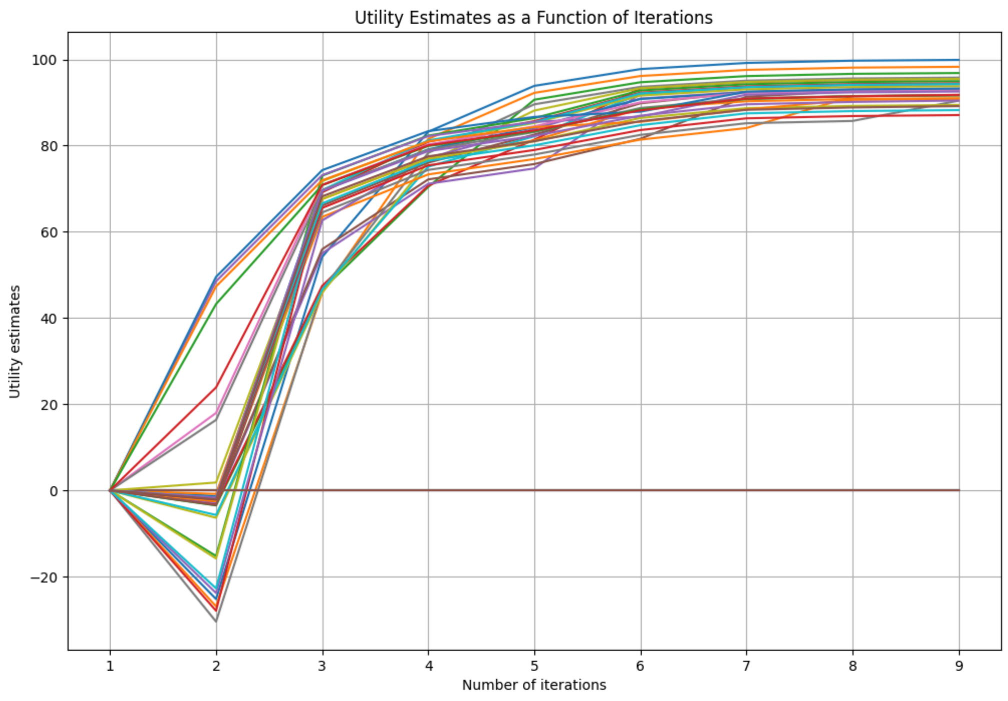


Figure 22. [Policy Iteration] Utility estimates as a function of the number of iterations

3.7 Finding optimal K value

Since the starting policy of each state is initialized randomly, we run the policy iteration algorithm 10 times.

K	Iterations to converge	Average Utility	Maximum Utility	Time to converge (seconds)
25	4 to 10	83.47 to 93.04	90.33 to 99.96	0.16 to 0.40
50	4 to 9	86.92 to 92.92	93.78 to 99.85	0.16 to 0.37
75	6 to 8	91.29 to 92.99	98.22 to 99.91	0.24 to 0.33
100	6 to 8	91.67 to 92.90	98.60 to 99.78	0.24 to 0.34
125	5 to 7	90.41 to 92.47	97.33 to 99.40	0.20 to 0.29

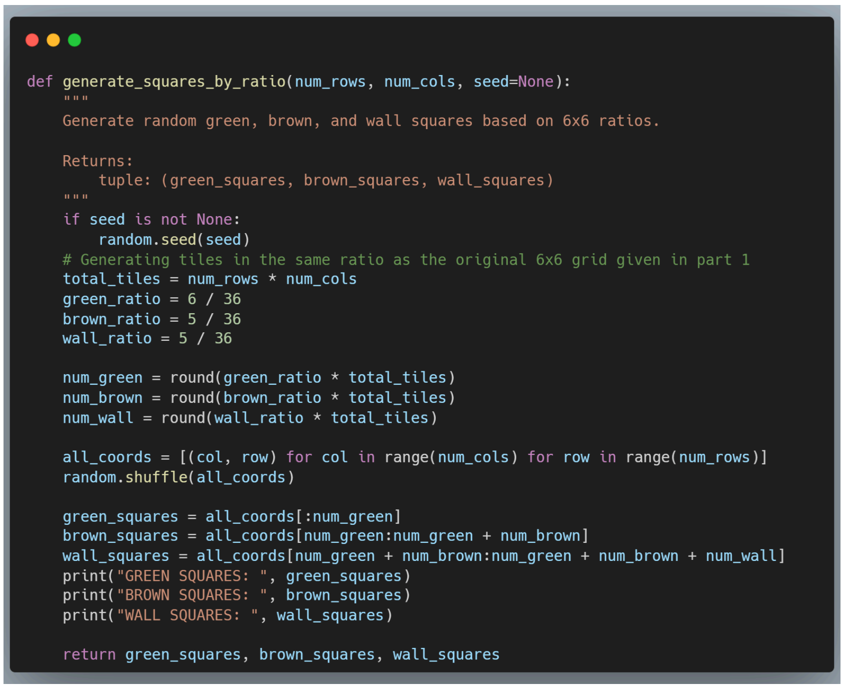
Table 1. K-Value Experiment for Policy Iteration (10 runs per K value)

Looking at the results in the table above, we choose K=100 to be the optimal K value as it provides the best balance of high max / avg utility values and reasonable time cost. Moreover, it consistently achieves the highest maximum utility values while maintaining stable convergence behavior..

4. Design a More Complicated Maze Environment

In this section, we complicate the maze environment by some factors which include increasing the dimensions of the grid to 6x6, 10x10, 20x20, 50x50, 100x100, 500x500, 1000x1000

1. Keep the convergence threshold constant across all grid sizes
2. (For Value Iteration only) Scale the convergence threshold according to the grid size



```python
def generate_squares_by_ratio(num_rows, num_cols, seed=None):
    """
    Generate random green, brown, and wall squares based on 6x6 ratios.

    Returns:
        tuple: (green_squares, brown_squares, wall_squares)
    """
    if seed is not None:
        random.seed(seed)
    # Generating tiles in the same ratio as the original 6x6 grid given in part 1
    total_tiles = num_rows * num_cols
    green_ratio = 6 / 36
    brown_ratio = 5 / 36
    wall_ratio = 5 / 36

    num_green = round(green_ratio * total_tiles)
    num_brown = round(brown_ratio * total_tiles)
    num_wall = round(wall_ratio * total_tiles)

    all_coords = [(col, row) for col in range(num_cols) for row in range(num_rows)]
    random.shuffle(all_coords)

    green_squares = all_coords[:num_green]
    brown_squares = all_coords[num_green:num_green + num_brown]
    wall_squares = all_coords[num_green + num_brown:num_green + num_brown + num_wall]
    print("GREEN SQUARES: ", green_squares)
    print("BROWN SQUARES: ", brown_squares)
    print("WALL SQUARES: ", wall_squares)

    return green_squares, brown_squares, wall_squares
```

Figure 23. Code snippet to randomly generate tiles in larger grid

The application is able to generate mazes of any dimensions as long as one configures it in the config.py file. The ratio of green tiles : brown tiles : walls is kept constant throughout the various grid sizes. We observed different behavior when the convergence threshold is kept constant and when we scale it according to the grid size. Both are discussed below.

4.1 Keep the convergence threshold constant

The parameter settings here is the same as in part 1 i.e.for value iteration: γ = 0.99 and c=0.05 and for policy iteration K=100

Grid Size	Value Iteration		Policy Iteration	
	# Iterations to Converge	Time taken to converge (seconds)	# Iterations to Converge	Time taken to converge (seconds)
6x6	757	0.73	4	0.1626
10x10	754	2.00	11	1.2513
20x20	757	8.54	11	5.1312
50x50	757	55.62	14	45.5883
100x100	757	234.14	16	217.7585
500x500	NA	> 600	NA	>600

Table 02 Evaluation results

For the case of value iteration, we observe that the number of iterations is constant and the time taken to converge increases quadratically.

A possible explanation for the constant iteration count could be that value iteration updates state values iteratively until the maximum change in value falls below the threshold. With a fixed threshold, the algorithm needs to reach the same level of precision for all grid sizes, which explains the consistent 917 iterations. And for the increase in time complexity, the computational complexity per iteration is $O(n^2)$, making the total time complexity $O(k \cdot n^2)$ where k is the iteration count. This explains why the 100×100 grid takes approximately 344× longer than the 6×6 grid despite using the same number of iterations.

Since the 500x500 grid took > 10 minutes to converge, we do not experiment with grid sizes larger than making it the upper threshold for obtaining the right policy. A possible solution to obtain a policy for larger grid sizes could be to use “finite time horizon” Using a finite time horizon would limit the number of steps the algorithm looks ahead, potentially making larger grid sizes tractable at the cost of some optimality guarantees.

On the other hand, for Policy Iteration, we observe that it makes more informed updates by evaluating complete policy at each step. It requires significantly fewer iterations (4 to 16) and scales sub-linearly with grid size.

4.2 Scale the convergence threshold according to the grid size

Another change that is implemented here is to scale the convergence threshold with grid size:

$$convergence\ threshold\ =\ \varepsilon\ *\ (\frac{1-\gamma}{\gamma})/(grid\ size)$$

Scaling ε dynamically with grid size ensures consistent convergence behavior across different grid sizes. In larger grids, utility updates diffuse more gradually, so a fixed ε might cause premature termination, leading to suboptimal policies. This makes intuitive sense because as grid size increases, the "signal" of value updates must travel farther, and smaller thresholds ensure these distant effects are properly accounted for.

Grid Size	Value Iteration	
	# Iterations to Converge	Time taken to converge (seconds)
6x6	1113	1.82
10x10	1212	5.59
20x20	1353	18.69
50x50	1695	137.6840
100x100	1833	618.4540
500x500	NA	>600

Table 03 Evaluation results

Based on the results, we see that it decreased the efficiency in terms of number of iteration and time complexity. For instance, the number of iterations runtime for 100x100 grids doubles. While the scaled threshold ensures more precise convergence, the computational cost must be taken into consideration. The threshold scaling factor $\frac{1}{grid\ size}$ might be too aggressive. A less strict scaling (e.g., $\frac{1}{\sqrt{grid\ size}}$) could balance precision and computational cost.