CE/CZ 4046: Intelligent Agents

Assignment 1: Agent Decision Making

INGALE OMKAR
(U2020724H)

School of Computer Science and Engineering
Nanyang Technological University, Singapore

TABLE OF CONTENTS

1. INTRODUCTION

Intelligent agents are autonomous systems that perceive their environment and take actions to achieve their goals. A crucial capability for these agents is effective decision-making. In the field of Artificial Intelligence, agent decision making refers to the design of algorithms that enable agents to select the best course of action within a given environment. This process considers the agent's current state, available actions, potential outcomes, and the associated rewards or penalties. This assignment deals with the challenge of an agent navigating a maze environment and choosing a path that aims to maximize its rewards. The given maze can be seen below.

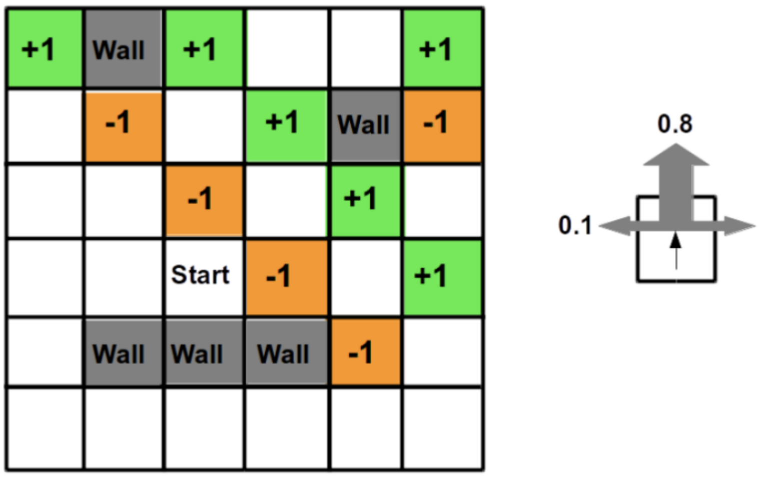


Figure 1: Maze Environment and Transition Probabilities

The environment is such that there is an 80% likelihood that an intended action occurs. There is another 10% likelihood that the agent moves to the right of the intended direction and another 10% likelihood that the agent moves to the left of the intended direction. If the agents proceeds to move (“bump”) into a wall, the action will not be carried out and the agent remains stationary in the current cell. The white (“empty”) cells have a reward of -0.04, the green cells have a reward of +1 and the brown cells have a reward of -1. Because there are no terminal states in the environment, the sequence of states that the agent can be in is limitless.

To solve this challenge and find the optimal policy and utilities of all the cells, we make use of value and policy iteration. Furthermore, the utility of cells with respect to the number of iterations is also plotted for further analysis.

2. Source Code Explanation

The source code accompanying this report is has been written entirely in Python 3. This programming language was used because of the plethora of in built functions and package support that made tasks such as printing out the maze a lot easier.

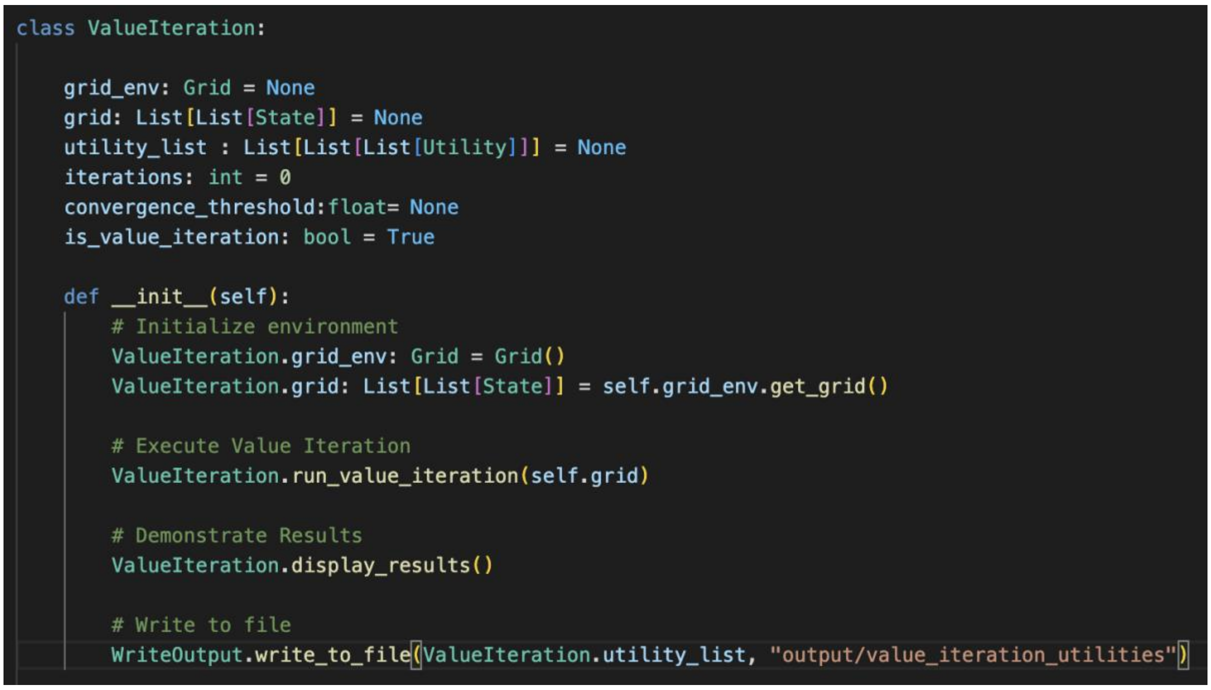
The following steps can be taken to execute the code:

1. Clone the repository.
2. Create a new python environment. This can be done using anaconda or any other package manager.
3. Install all the dependencies mentioned in the “requirements.txt” file.
4. Run/Execute the files: “value_iteration.py”, “policy_iteration.py” or “complex_maze.py” for results on value iteration, policy iteration and the complex maze task respectively.
5. You can alter the constants such as the c value or the k value in the iteration_constant.py file.

The following table elaborates on the function of each file in the codebase except for value_iteration.py, policy_iteration.py and complex_maze.py as those files implement the algorithms for the respective questions:

state.py	Represents the state of a grid cell. Can be reward value or wall
utility.py	Represents the utility of a state, given an action
grid.py	Represents the grid environment with states
action.py	Represents different actions that can be taken: up, down, left, right
grid_constant.py	Lists constant values that are related to the grid (rows, cols, etc)
iteration_constant.py	Lists all the constant values that are related to the iterations (c, k, epsilon, etc)
show_grid.py	Has functions to help with printing the grid with the help of the tabulate python package
write_output.py	Helps write the data to csv files
utility_controller.py	Calculate best utility in current state and estimate best utility in next policy iteration

3. Value Iteration Implementation



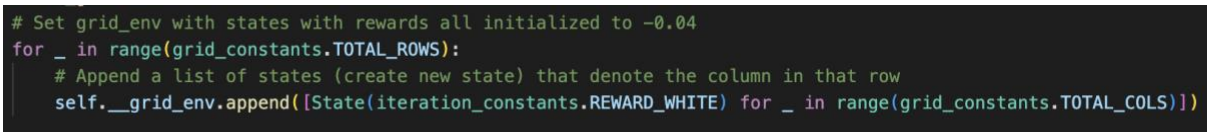
```
class ValueIteration:

    grid_env: Grid = None
    grid: List[List[State]] = None
    utility_list : List[List[List[Utility]]] = None
    iterations: int = 0
    convergence_threshold:float= None
    is_value_iteration: bool = True

    def __init__(self):
        # Initialize environment
        ValueIteration.grid_env: Grid = Grid()
        ValueIteration.grid: List[List[State]] = self.grid_env.get_grid()

        # Execute Value Iteration
        ValueIteration.run_value_iteration(self.grid)

        # Demonstrate Results
        ValueIteration.display_results()

        # Write to file
        WriteOutput.write_to_file(ValueIteration.utility_list, "output/value_iteration_utilities")
```

Figure 2: Value Iteration Implementation

The Value Iteration Algorithm was implemented as a class. With key functions tagged as "@staticmethods", the entire algorithm works by creating a new ValueIteration class instance from the " __main__" method. This class builds everything from the ground up using other defined classes and constant values. This section walks through the complete initialisation of the ValueIteration object as defined in its constructor and how the algorithm is implemented.

3.1 Initialisation of the Grid

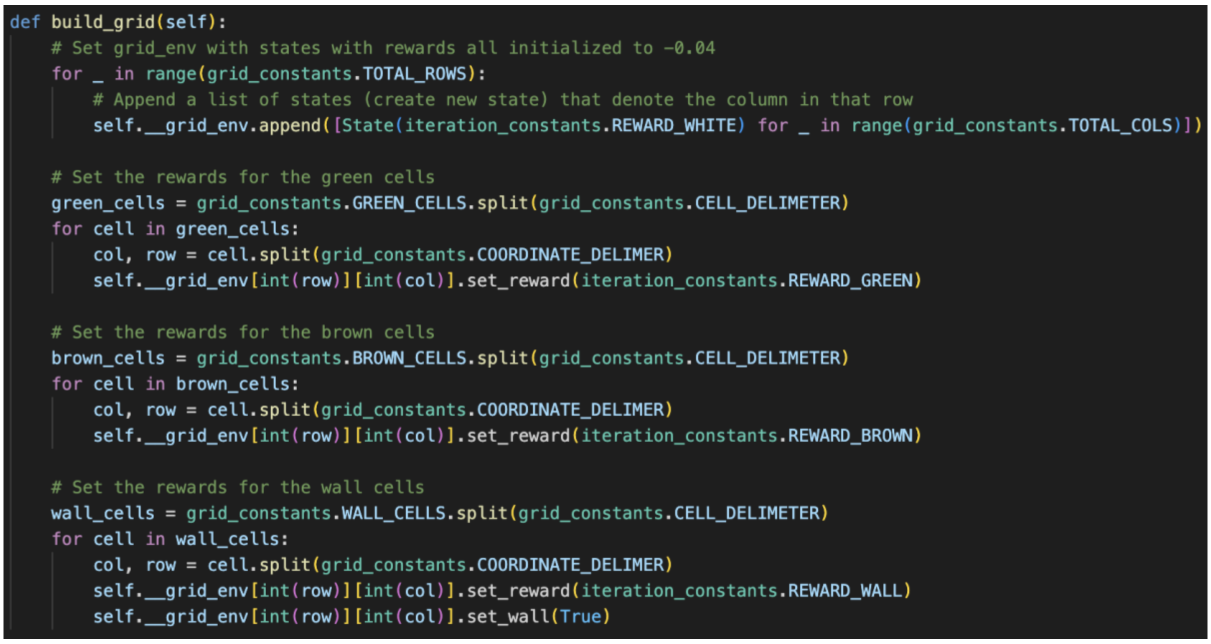
Before the value iteration can be run, a grid needs to be created. This is done using the Grid class. Grid, in this program, is defined as a 2d array of State classes. So, when a grid is initialised, it is mapped to a 2d array of States and given a reward value of White cells (-0.04). This is done for all the cells in the grid.



```
# Set grid_env with states with rewards all initialized to -0.04
for _ in range(grid_constants.TOTAL_ROWS):
    # Append a list of states (create new state) that denote the column in that row
    self.__grid_env.append([State(iteration_constants.REWARD_WHITE) for _ in range(grid_constants.TOTAL_COLS)])
```

Figure 3: Initialising the cells in the grid

After all the cells in the grid have been initialised, the lists of cells corresponding to green cells, brown cells, and walls is retrieved from the declared constant values. These cells are then identified in the grid and their reward values are set according to the corresponding constant values that are also declared in the grid constants file.

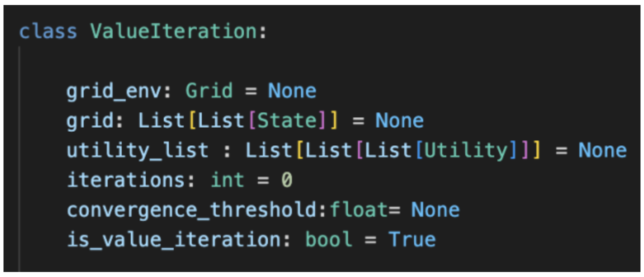


```python
def build_grid(self):
    # Set grid_env with states with rewards all initialized to -0.04
    for _ in range(grid_constants.TOTAL_ROWS):
        # Append a list of states (create new state) that denote the column in that row
        self.__grid_env.append([State(iteration_constants.REWARD_WHITE) for _ in range(grid_constants.TOTAL_COLS)])

    # Set the rewards for the green cells
    green_cells = grid_constants.GREEN_CELLS.split(grid_constants.CELL_DELIMETER)
    for cell in green_cells:
        col, row = cell.split(grid_constants.COORDINATE_DELIMER)
        self.__grid_env[int(row)][int(col)].set_reward(iteration_constants.REWARD_GREEN)

    # Set the rewards for the brown cells
    brown_cells = grid_constants.BROWN_CELLS.split(grid_constants.CELL_DELIMETER)
    for cell in brown_cells:
        col, row = cell.split(grid_constants.COORDINATE_DELIMER)
        self.__grid_env[int(row)][int(col)].set_reward(iteration_constants.REWARD_BROWN)

    # Set the rewards for the wall cells
    wall_cells = grid_constants.WALL_CELLS.split(grid_constants.CELL_DELIMETER)
    for cell in wall_cells:
        col, row = cell.split(grid_constants.COORDINATE_DELIMER)
        self.__grid_env[int(row)][int(col)].set_reward(iteration_constants.REWARD_WALL)
        self.__grid_env[int(row)][int(col)].set_wall(True)
```

Figure 4: The entire grid building method

3.2 Value Iteration for Finding Optimal Policy

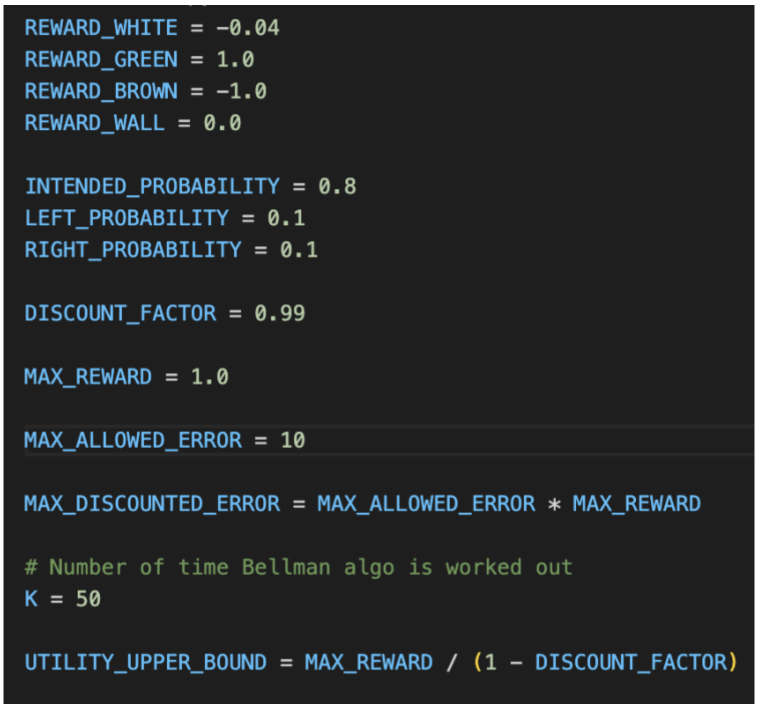
For finding the optimal policy using value iteration, two utility vectors are created. The purpose of these two utility vectors is to store the utility of states, both current and updated, after every iteration. All the utilities in these utility vectors are initialised as 0.0. Furthermore, another list is kept to store all the utility states from every iteration. This is done to map the change in state utility values after every iteration. Another variable that is used in this is delta. This variable is initialised to be the least possible negative float value. Delta's value is updated when a new delta that is larger in magnitude compared to the previous value is calculated. As there are no terminal states for value iteration, the sequence of states could possibly extend to an infinite chain. To deal with this, a convergence threshold is calculated and used as a condition to terminate the process.



```
class ValueIteration:

    grid_env: Grid = None
    grid: List[List[State]] = None
    utility_list : List[List[List[Utility]]] = None
    iterations: int = 0
    convergence_threshold:float= None
    is_value_iteration: bool = True
```

Figure 5: Variables used in Value Iteration

The maximum allowed error in a state (also known as Epsilon) is also defined in the iteration constants file. This value is crucial as variance in this value also results in different optimal policies being yielded by the algorithm. Likewise, other crucial values such as discount factor, upper utility bound, etc are also described in the iterations constant file.



```
REWARD_WHITE = -0.04
REWARD_GREEN = 1.0
REWARD_BROWN = -1.0
REWARD_WALL = 0.0

INTENDED_PROBABILITY = 0.8
LEFT_PROBABILITY = 0.1
RIGHT_PROBABILITY = 0.1

DISCOUNT_FACTOR = 0.99

MAX_REWARD = 1.0

MAX_ALLOWED_ERROR = 10

MAX_DISCOUNTED_ERROR = MAX_ALLOWED_ERROR * MAX_REWARD

# Number of time Bellman algo is worked out
K = 50

UTILITY_UPPER_BOUND = MAX_REWARD / (1 - DISCOUNT_FACTOR)
```

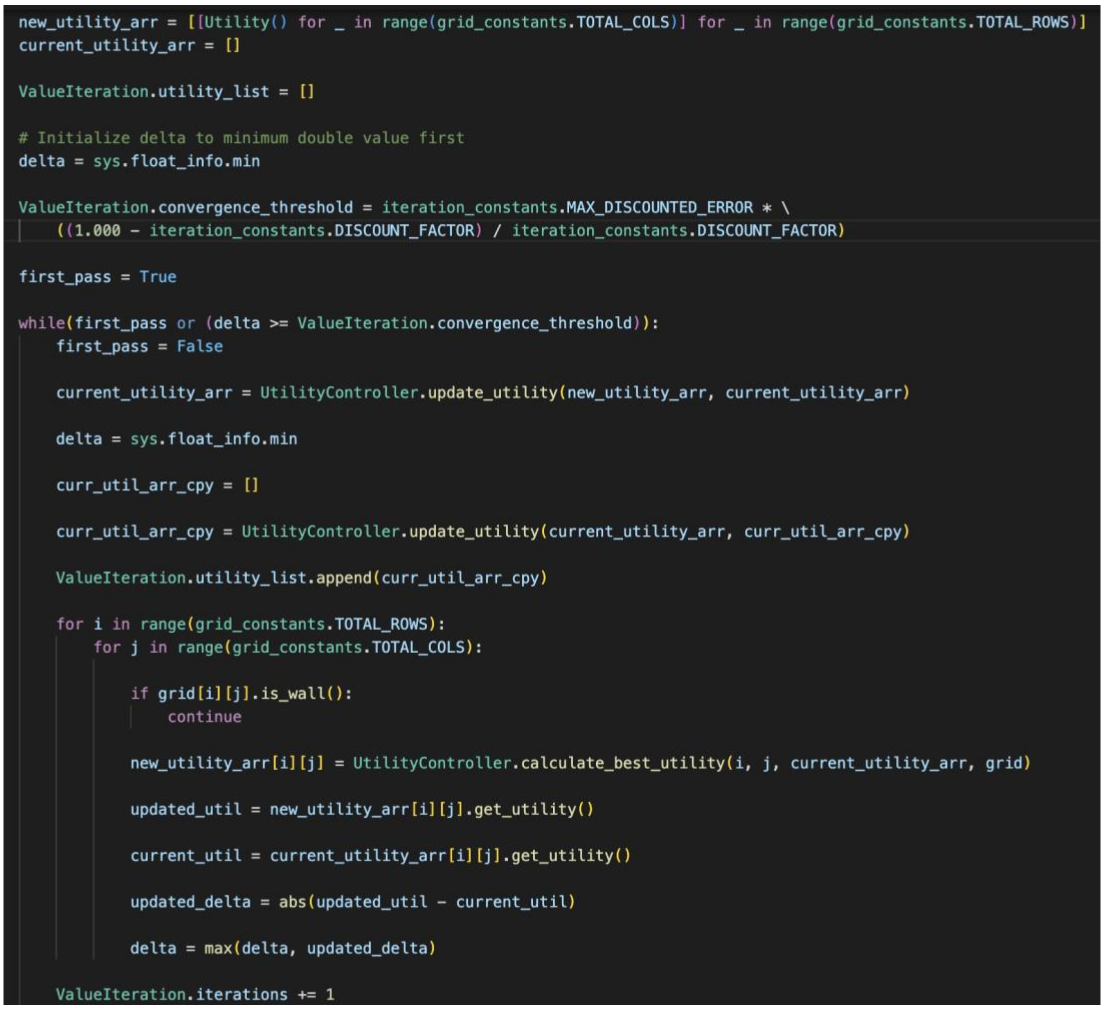
Figure 6: Constants as defined in iteration constants

As mentioned before, the maze or the grid has no terminal states. This means that the sequence of states could possibly never terminate. In such cases, there is no “deadline” that is imposed when trying to determine the optimal policy. What this also implies is that the agent’s starting state does not affect the optimal policy that is determined later. Hence, the agent’s starting position in the grid will have no impact on final optimal policy.

During each iteration, the current utility vector that we initialised at the start of the algorithm will be updated to carry the best found utility till this iteration. After this, the vector carries the estimated utility values for all states. A copy of this vector is made and stored in the utility list that helps keep track of utilities of all states in every iteration. This is done so we can express the utility of states as a function of iteration after the optimal policy has been found.

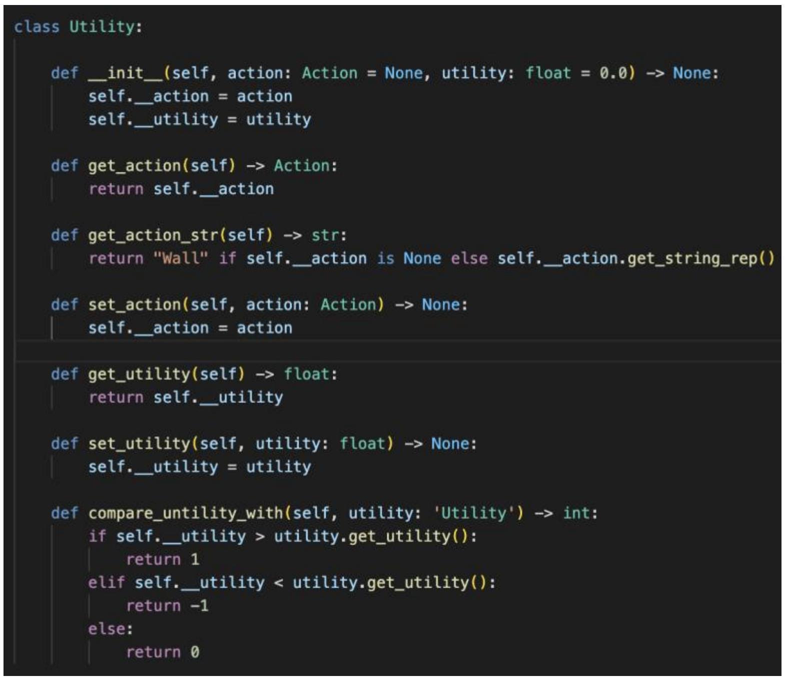
After that, the best utility for each state is calculated by choosing the action that will maximize the expected utility of the subsequent state. In the implementation, this is done by creating a list of utilities corresponding to the actions that produce them. Given an action (example: up), the utility is calculated and placed on the list. The list is then sorted using a lambda function to find the highest utility value and update the new best utility value for a state.

Lastly, for each state, the delta (difference between new best utility and the current utility of the state) is calculated. If the new delta value exceeds the current delta value, the current value will be updated and replaced with the new one. This is done for all the states in the environment, post which the program checks if the current delta value is lower in magnitude compared to the convergence threshold. This is where the “terminating” condition for the loop is. If the current delta value is lower than the convergence threshold, the loop will terminate and the optimal policy would be finalised. A logistical conclusion of this threshold is that the change in delta after this point will be negligible compared to the maximum error allowed. This is an essential check to ensure that the loop does not continue infinitely.



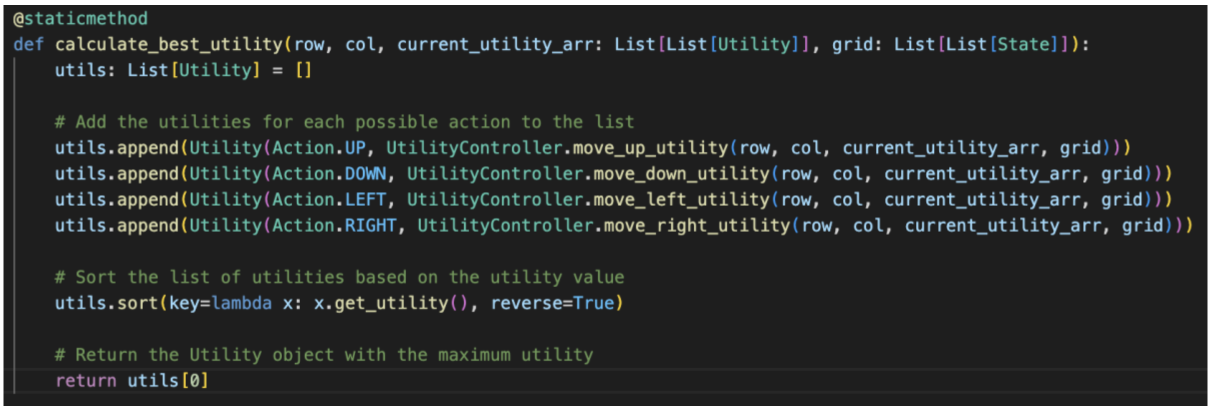
```python
new_utility_arr = [[Utility() for _ in range(grid_constants.TOTAL_COLS)] for _ in range(grid_constants.TOTAL_ROWS)]
current_utility_arr = []

ValueIteration.utility_list = []

# Initialize delta to minimum double value first
delta = sys.float_info.min

ValueIteration.convergence_threshold = iteration_constants.MAX_DISCOUNTED_ERROR * \
    ((1.000 - iteration_constants.DISCOUNT_FACTOR) / iteration_constants.DISCOUNT_FACTOR)

first_pass = True

while(first_pass or (delta >= ValueIteration.convergence_threshold)):
    first_pass = False

    current_utility_arr = UtilityController.update_utility(new_utility_arr, current_utility_arr)

    delta = sys.float_info.min

    curr_util_arr_cpy = []

    curr_util_arr_cpy = UtilityController.update_utility(current_utility_arr, curr_util_arr_cpy)

    ValueIteration.utility_list.append(curr_util_arr_cpy)

    for i in range(grid_constants.TOTAL_ROWS):
        for j in range(grid_constants.TOTAL_COLS):

            if grid[i][j].is_wall():
                continue

            new_utility_arr[i][j] = UtilityController.calculate_best_utility(i, j, current_utility_arr, grid)

            updated_util = new_utility_arr[i][j].get_utility()

            current_util = current_utility_arr[i][j].get_utility()

            updated_delta = abs(updated_util - current_util)

            delta = max(delta, updated_delta)

    ValueIteration.iterations += 1
```

Figure 8: Implementation of the Run Value Iteration Method



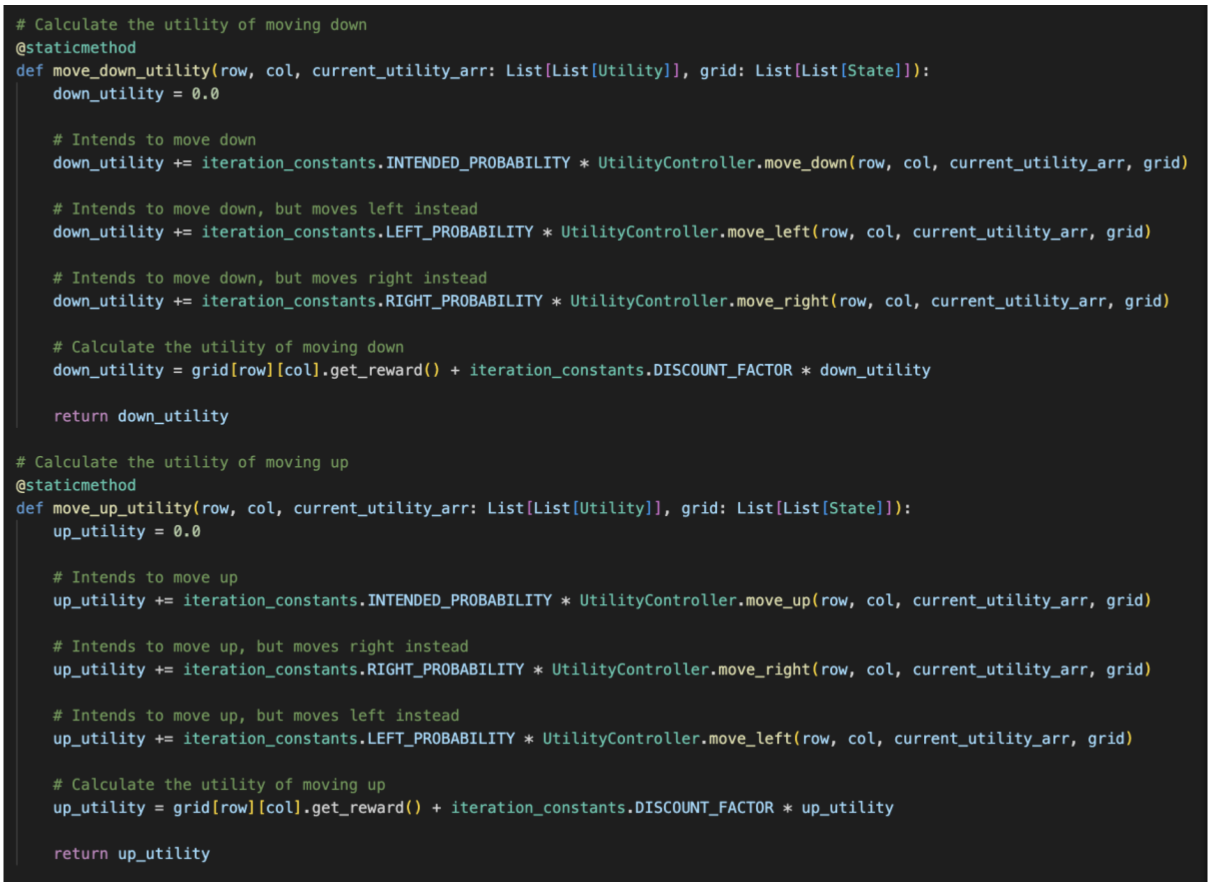
```python
class Utility:

    def __init__(self, action: Action = None, utility: float = 0.0) -> None:
        self.__action = action
        self.__utility = utility

    def get_action(self) -> Action:
        return self.__action

    def get_action_str(self) -> str:
        return "Wall" if self.__action is None else self.__action.get_string_rep()

    def set_action(self, action: Action) -> None:
        self.__action = action

    def get_utility(self) -> float:
        return self.__utility

    def set_utility(self, utility: float) -> None:
        self.__utility = utility

    def compare_untility_with(self, utility: 'Utility') -> int:
        if self.__utility > utility.get_utility():
            return 1
        elif self.__utility < utility.get_utility():
            return -1
        else:
            return 0
```

Figure 9: Implementation of the Utility Class



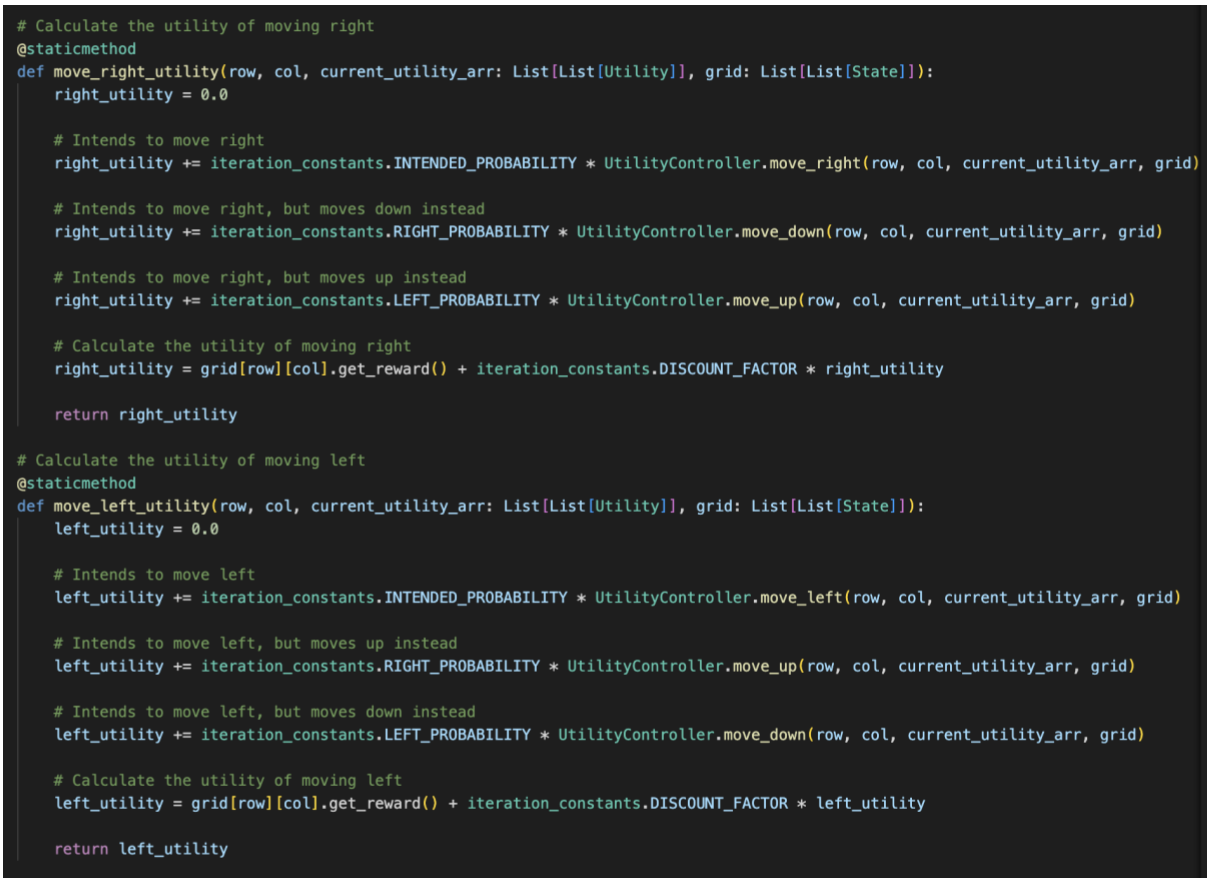
```python
@staticmethod
def calculate_best_utility(row, col, current_utility_arr: List[List[Utility]], grid: List[List[State]]):
    utils: List[Utility] = []

    # Add the utilities for each possible action to the list
    utils.append(Utility(Action.UP, UtilityController.move_up_utility(row, col, current_utility_arr, grid)))
    utils.append(Utility(Action.DOWN, UtilityController.move_down_utility(row, col, current_utility_arr, grid)))
    utils.append(Utility(Action.LEFT, UtilityController.move_left_utility(row, col, current_utility_arr, grid)))
    utils.append(Utility(Action.RIGHT, UtilityController.move_right_utility(row, col, current_utility_arr, grid)))

    # Sort the list of utilities based on the utility value
    utils.sort(key=lambda x: x.get_utility(), reverse=True)

    # Return the Utility object with the maximum utility
    return utils[0]
```

Figure 10: Update current utility vector with new best utility



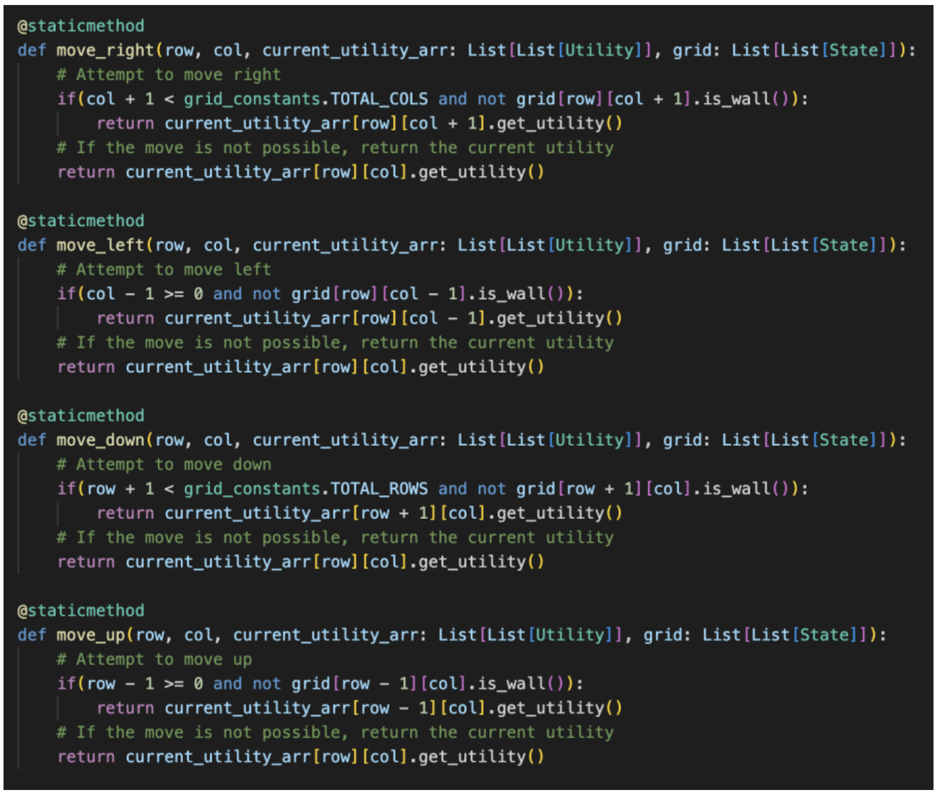
```python
# Calculate the utility of moving down
@staticmethod
def move_down_utility(row, col, current_utility_arr: List[List[Utility]], grid: List[List[State]]):
    down_utility = 0.0

    # Intends to move down
    down_utility += iteration_constants.INTENDED_PROBABILITY * UtilityController.move_down(row, col, current_utility_arr, grid)

    # Intends to move down, but moves left instead
    down_utility += iteration_constants.LEFT_PROBABILITY * UtilityController.move_left(row, col, current_utility_arr, grid)

    # Intends to move down, but moves right instead
    down_utility += iteration_constants.RIGHT_PROBABILITY * UtilityController.move_right(row, col, current_utility_arr, grid)

    # Calculate the utility of moving down
    down_utility = grid[row][col].get_reward() + iteration_constants.DISCOUNT_FACTOR * down_utility

    return down_utility

# Calculate the utility of moving up
@staticmethod
def move_up_utility(row, col, current_utility_arr: List[List[Utility]], grid: List[List[State]]):
    up_utility = 0.0

    # Intends to move up
    up_utility += iteration_constants.INTENDED_PROBABILITY * UtilityController.move_up(row, col, current_utility_arr, grid)

    # Intends to move up, but moves right instead
    up_utility += iteration_constants.RIGHT_PROBABILITY * UtilityController.move_right(row, col, current_utility_arr, grid)

    # Intends to move up, but moves left instead
    up_utility += iteration_constants.LEFT_PROBABILITY * UtilityController.move_left(row, col, current_utility_arr, grid)

    # Calculate the utility of moving up
    up_utility = grid[row][col].get_reward() + iteration_constants.DISCOUNT_FACTOR * up_utility

    return up_utility
```

Figure 11: Calculating utility for movement along vertical axis



```python
# Calculate the utility of moving right
@staticmethod
def move_right_utility(row, col, current_utility_arr: List[List[Utility]], grid: List[List[State]]):
    right_utility = 0.0

    # Intends to move right
    right_utility += iteration_constants.INTENDED_PROBABILITY * UtilityController.move_right(row, col, current_utility_arr, grid)

    # Intends to move right, but moves down instead
    right_utility += iteration_constants.RIGHT_PROBABILITY * UtilityController.move_down(row, col, current_utility_arr, grid)

    # Intends to move right, but moves up instead
    right_utility += iteration_constants.LEFT_PROBABILITY * UtilityController.move_up(row, col, current_utility_arr, grid)

    # Calculate the utility of moving right
    right_utility = grid[row][col].get_reward() + iteration_constants.DISCOUNT_FACTOR * right_utility

    return right_utility

# Calculate the utility of moving left
@staticmethod
def move_left_utility(row, col, current_utility_arr: List[List[Utility]], grid: List[List[State]]):
    left_utility = 0.0

    # Intends to move left
    left_utility += iteration_constants.INTENDED_PROBABILITY * UtilityController.move_left(row, col, current_utility_arr, grid)

    # Intends to move left, but moves up instead
    left_utility += iteration_constants.RIGHT_PROBABILITY * UtilityController.move_up(row, col, current_utility_arr, grid)

    # Intends to move left, but moves down instead
    left_utility += iteration_constants.LEFT_PROBABILITY * UtilityController.move_down(row, col, current_utility_arr, grid)

    # Calculate the utility of moving left
    left_utility = grid[row][col].get_reward() + iteration_constants.DISCOUNT_FACTOR * left_utility

    return left_utility
```

Figure 12: Calculating utility for movement along the horizontal axis



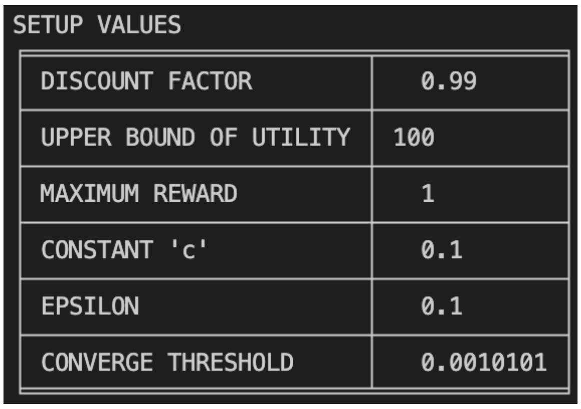
```python
@staticmethod
def move_right(row, col, current_utility_arr: List[List[Utility]], grid: List[List[State]]):
    # Attempt to move right
    if(col + 1 < grid_constants.TOTAL_COLS and not grid[row][col + 1].is_wall()):
        return current_utility_arr[row][col + 1].get_utility()
    # If the move is not possible, return the current utility
    return current_utility_arr[row][col].get_utility()

@staticmethod
def move_left(row, col, current_utility_arr: List[List[Utility]], grid: List[List[State]]):
    # Attempt to move left
    if(col - 1 >= 0 and not grid[row][col - 1].is_wall()):
        return current_utility_arr[row][col - 1].get_utility()
    # If the move is not possible, return the current utility
    return current_utility_arr[row][col].get_utility()

@staticmethod
def move_down(row, col, current_utility_arr: List[List[Utility]], grid: List[List[State]]):
    # Attempt to move down
    if(row + 1 < grid_constants.TOTAL_ROWS and not grid[row + 1][col].is_wall()):
        return current_utility_arr[row + 1][col].get_utility()
    # If the move is not possible, return the current utility
    return current_utility_arr[row][col].get_utility()

@staticmethod
def move_up(row, col, current_utility_arr: List[List[Utility]], grid: List[List[State]]):
    # Attempt to move up
    if(row - 1 >= 0 and not grid[row - 1][col].is_wall()):
        return current_utility_arr[row - 1][col].get_utility()
    # If the move is not possible, return the current utility
    return current_utility_arr[row][col].get_utility()
```

Figure 13: Calculating utility for taking ACTION along both axes

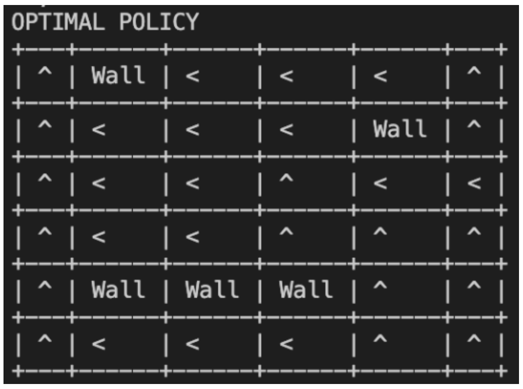
3.3 Plot for Optimal Policy and Different “c” Values

The following section highlights different cases with different “c” values. All of the data provided will be in the form of images taking directly from the terminal after execution.

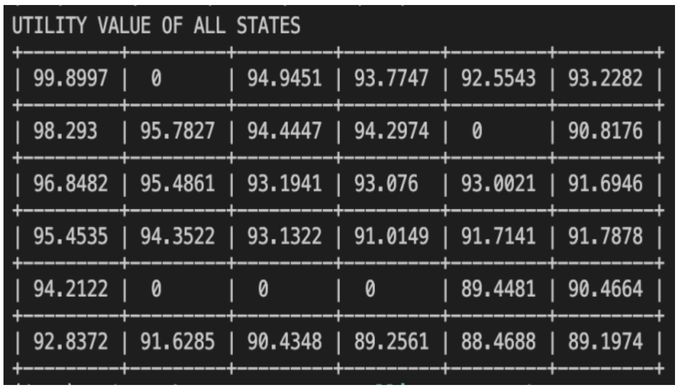
3.3.1 C = 0.1



```
SETUP VALUES
+-----------------------+-----------+
| DISCOUNT FACTOR       | 0.99      |
| UPPER BOUND OF UTILITY| 100       |
| MAXIMUM REWARD        | 1         |
| CONSTANT 'c'          | 0.1       |
| EPSILON               | 0.1       |
| CONVERGE THRESHOLD    | 0.0010101 |
+-----------------------+-----------+
```

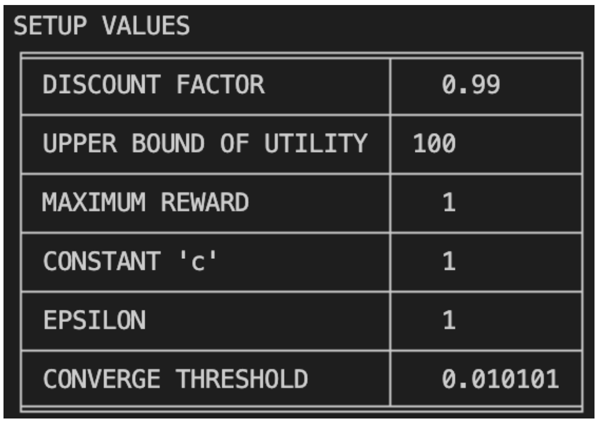


```
OPTIMAL POLICY
+---+------+------+------+------+---+
| ^ | Wall | <    | <    | <    | ^ |
+---+------+------+------+------+---+
| ^ | <    | <    | <    | Wall | ^ |
+---+------+------+------+------+---+
| ^ | <    | <    | ^    | <    | < |
+---+------+------+------+------+---+
| ^ | <    | <    | ^    | ^    | ^ |
+---+------+------+------+------+---+
| ^ | Wall | Wall | Wall | ^    | ^ |
+---+------+------+------+------+---+
| ^ | <    | <    | <    | ^    | ^ |
+---+------+------+------+------+---+
```

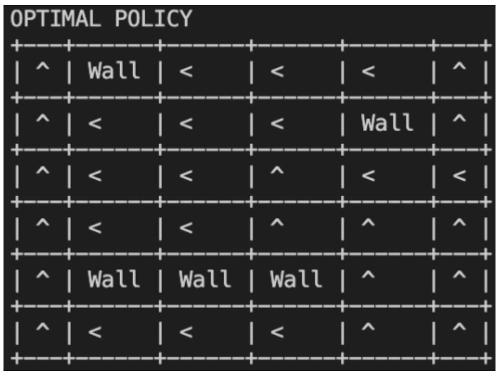


```
UTILITY VALUE OF ALL STATES
+---------+---------+---------+---------+---------+---------+
| 99.8997 | 0       | 94.9451 | 93.7747 | 92.5543 | 93.2282 |
+---------+---------+---------+---------+---------+---------+
| 98.293  | 95.7827 | 94.4447 | 94.2974 | 0       | 90.8176 |
+---------+---------+---------+---------+---------+---------+
| 96.8482 | 95.4861 | 93.1941 | 93.076  | 93.0021 | 91.6946 |
+---------+---------+---------+---------+---------+---------+
| 95.4535 | 94.3522 | 93.1322 | 91.0149 | 91.7141 | 91.7878 |
+---------+---------+---------+---------+---------+---------+
| 94.2122 | 0       | 0       | 0       | 89.4481 | 90.4664 |
+---------+---------+---------+---------+---------+---------+
| 92.8372 | 91.6285 | 90.4348 | 89.2561 | 88.4688 | 89.1974 |
+---------+---------+---------+---------+---------+---------+
```

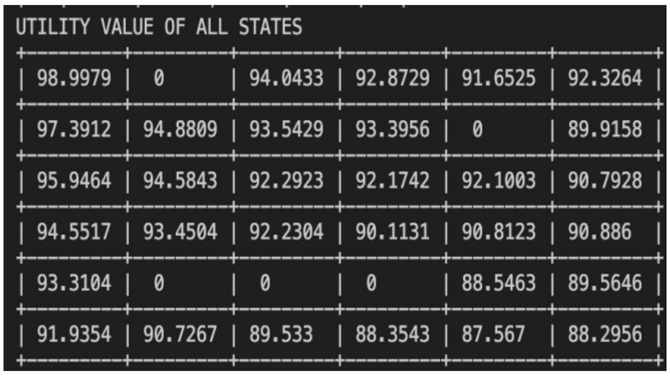
3.3.2 C = 1.0



```
SETUP VALUES
+-----------------------+-----------+
| DISCOUNT FACTOR       | 0.99      |
| UPPER BOUND OF UTILITY| 100       |
| MAXIMUM REWARD        | 1         |
| CONSTANT 'c'          | 1         |
| EPSILON               | 1         |
| CONVERGE THRESHOLD    | 0.010101  |
+-----------------------+-----------+
```

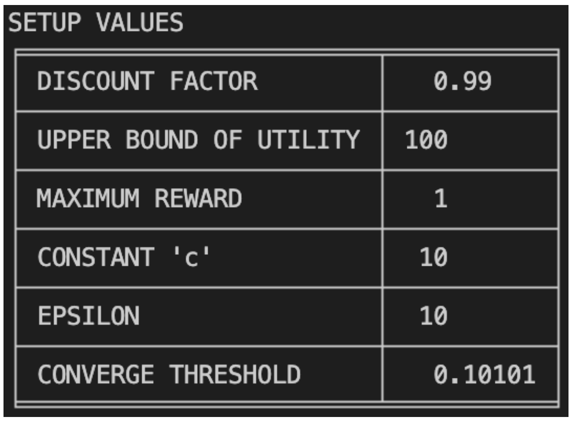


```
OPTIMAL POLICY
+---+------+------+------+------+---+
| ^ | Wall | <    | <    | <    | ^ |
+---+------+------+------+------+---+
| ^ | <    | <    | <    | Wall | ^ |
+---+------+------+------+------+---+
| ^ | <    | <    | ^    | <    | < |
+---+------+------+------+------+---+
| ^ | <    | <    | ^    | ^    | ^ |
+---+------+------+------+------+---+
| ^ | Wall | Wall | Wall | ^    | ^ |
+---+------+------+------+------+---+
| ^ | <    | <    | <    | ^    | ^ |
+---+------+------+------+------+---+
```



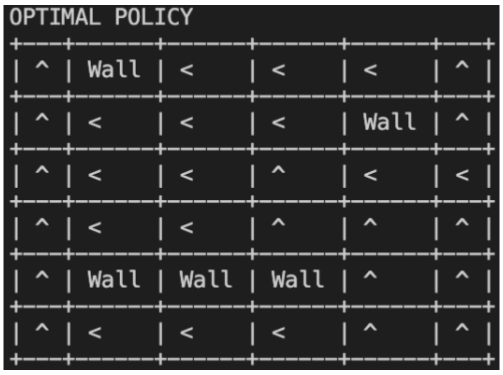
```
UTILITY VALUE OF ALL STATES
+---------+---------+---------+---------+---------+---------+
| 98.9979 |  0      | 94.0433 | 92.8729 | 91.6525 | 92.3264 |
+---------+---------+---------+---------+---------+---------+
| 97.3912 | 94.8809 | 93.5429 | 93.3956 |  0      | 89.9158 |
+---------+---------+---------+---------+---------+---------+
| 95.9464 | 94.5843 | 92.2923 | 92.1742 | 92.1003 | 90.7928 |
+---------+---------+---------+---------+---------+---------+
| 94.5517 | 93.4504 | 92.2304 | 90.1131 | 90.8123 | 90.886  |
+---------+---------+---------+---------+---------+---------+
| 93.3104 |  0      |  0      |  0      | 88.5463 | 89.5646 |
+---------+---------+---------+---------+---------+---------+
| 91.9354 | 90.7267 | 89.533  | 88.3543 | 87.567  | 88.2956 |
+---------+---------+---------+---------+---------+---------+
```

3.3.3 C = 10.0

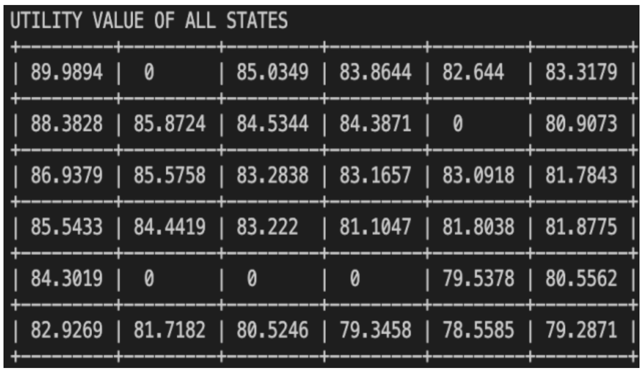


SETUP VALUES

DISCOUNT FACTOR	0.99
UPPER BOUND OF UTILITY	100
MAXIMUM REWARD	1
CONSTANT 'c'	10
EPSILON	10
CONVERGE THRESHOLD	0.10101

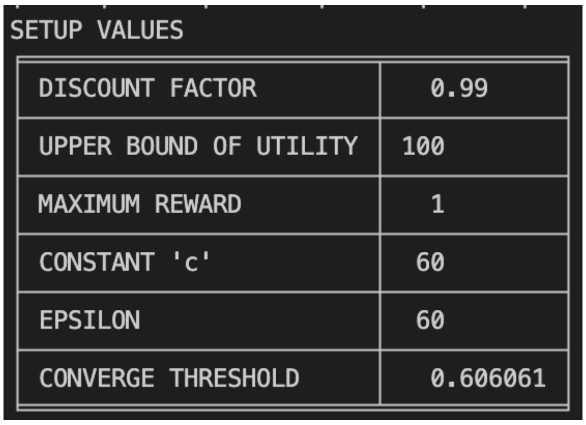


```
OPTIMAL POLICY
+---+------+------+------+------+---+
| ^ | Wall | <    | <    | <    | ^ |
+---+------+------+------+------+---+
| ^ | <    | <    | <    | Wall | ^ |
+---+------+------+------+------+---+
| ^ | <    | <    | ^    | <    | < |
+---+------+------+------+------+---+
| ^ | <    | <    | ^    | ^    | ^ |
+---+------+------+------+------+---+
| ^ | Wall | Wall | Wall | ^    | ^ |
+---+------+------+------+------+---+
| ^ | <    | <    | <    | ^    | ^ |
+---+------+------+------+------+---+
```

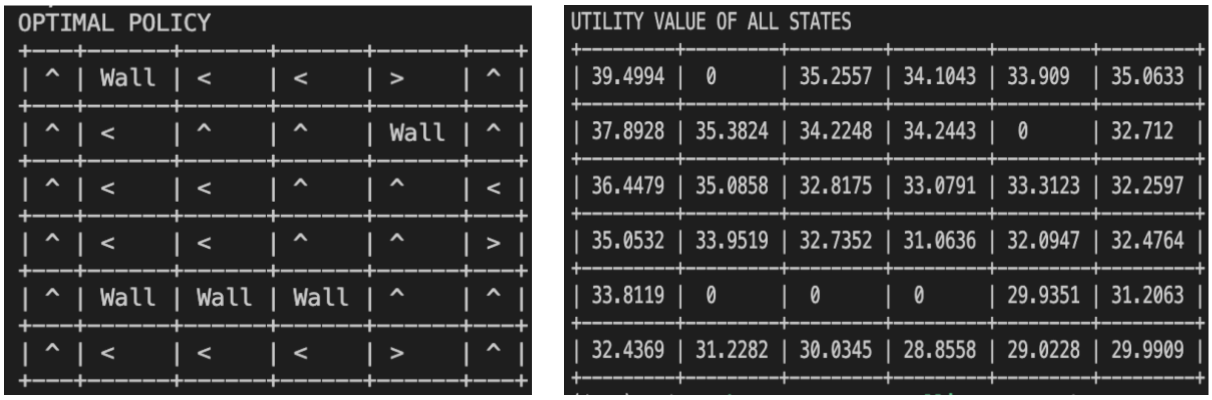


```
UTILITY VALUE OF ALL STATES
+---------+---------+---------+---------+---------+---------+
| 89.9894 |  0      | 85.0349 | 83.8644 | 82.644  | 83.3179 |
+---------+---------+---------+---------+---------+---------+
| 88.3828 | 85.8724 | 84.5344 | 84.3871 |  0      | 80.9073 |
+---------+---------+---------+---------+---------+---------+
| 86.9379 | 85.5758 | 83.2838 | 83.1657 | 83.0918 | 81.7843 |
+---------+---------+---------+---------+---------+---------+
| 85.5433 | 84.4419 | 83.222  | 81.1047 | 81.8038 | 81.8775 |
+---------+---------+---------+---------+---------+---------+
| 84.3019 |  0      |  0      |  0      | 79.5378 | 80.5562 |
+---------+---------+---------+---------+---------+---------+
| 82.9269 | 81.7182 | 80.5246 | 79.3458 | 78.5585 | 79.2871 |
+---------+---------+---------+---------+---------+---------+
```

3.3.4 C = 60



```
SETUP VALUES
+--------------------------+-----------+
| DISCOUNT FACTOR          | 0.99      |
+--------------------------+-----------+
| UPPER BOUND OF UTILITY   | 100       |
+--------------------------+-----------+
| MAXIMUM REWARD           | 1         |
+--------------------------+-----------+
| CONSTANT 'c'             | 60        |
+--------------------------+-----------+
| EPSILON                  | 60        |
+--------------------------+-----------+
| CONVERGE THRESHOLD       | 0.606061  |
+--------------------------+-----------+
```



```
OPTIMAL POLICY
+---+------+------+------+------+---+
| ^ | Wall | <    | <    | >    | ^ |
+---+------+------+------+------+---+
| ^ | <    | ^    | ^    | Wall | ^ |
+---+------+------+------+------+---+
| ^ | <    | <    | ^    | ^    | < |
+---+------+------+------+------+---+
| ^ | <    | <    | ^    | ^    | > |
+---+------+------+------+------+---+
| ^ | Wall | Wall | Wall | ^    | ^ |
+---+------+------+------+------+---+
| ^ | <    | <    | <    | >    | ^ |
+---+------+------+------+------+---+
```

```
UTILITY VALUE OF ALL STATES
+---------+---------+---------+---------+---------+---------+
| 39.4994 | 0       | 35.2557 | 34.1043 | 33.909  | 35.0633 |
+---------+---------+---------+---------+---------+---------+
| 37.8928 | 35.3824 | 34.2248 | 34.2443 | 0       | 32.712  |
+---------+---------+---------+---------+---------+---------+
| 36.4479 | 35.0858 | 32.8175 | 33.0791 | 33.3123 | 32.2597 |
+---------+---------+---------+---------+---------+---------+
| 35.0532 | 33.9519 | 32.7352 | 31.0636 | 32.0947 | 32.4764 |
+---------+---------+---------+---------+---------+---------+
| 33.8119 | 0       | 0       | 0       | 29.9351 | 31.2063 |
+---------+---------+---------+---------+---------+---------+
| 32.4369 | 31.2282 | 30.0345 | 28.8558 | 29.0228 | 29.9909 |
+---------+---------+---------+---------+---------+---------+
```

Figure 14-26: Output for the value iteration algorithm

One observation that can be made is that as the value of "c" increases, there are some changes in the optimal policy. This is evident in cell (4, 5) where the policy went from "^" (UP) to ">" (RIGHT).

The plot on the following page shows the relationship between utility value and iterations for c = 30. This value was chosen arbitrarily for demonstration purposes. If plots for other values are required, they can be made by executing data_exploration.ipynb file in the codebase.

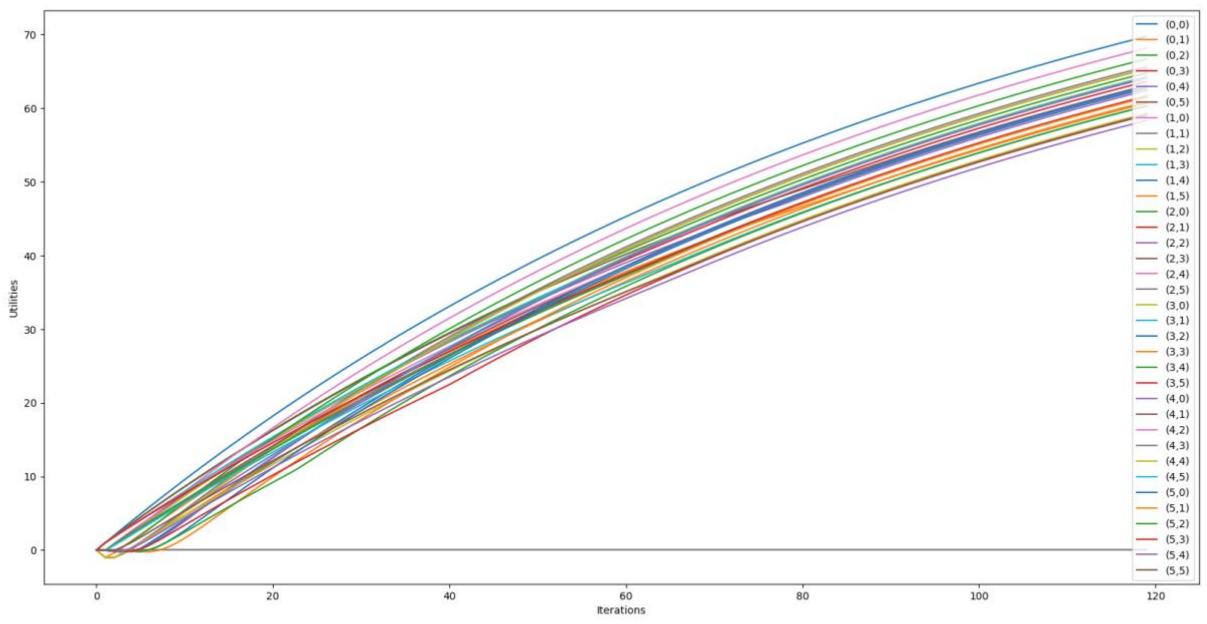


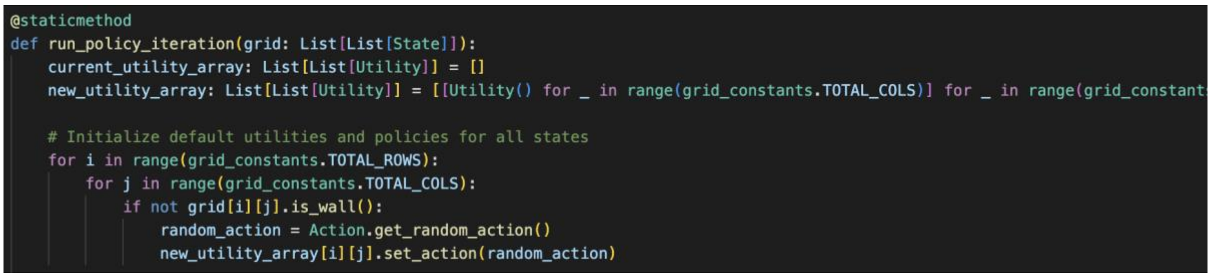
Figure 27: Utility value vs Iteration for C = 30

4. Policy Iteration Implementation

Policy iteration also follows the same program flow as value iteration. The only difference is that this time the policy_iteration.py file is executed.

4.1 Policy Iteration for Finding the Optimal Policy

This time, only one vector of current utilities is initialised to store the current utilities of states. However, another vector, a new utilities vector, indexed by state is created to store the updated utilities of states after each iteration. All the states in the policy vector will be initialized with utility values of 0.0 and a random action.



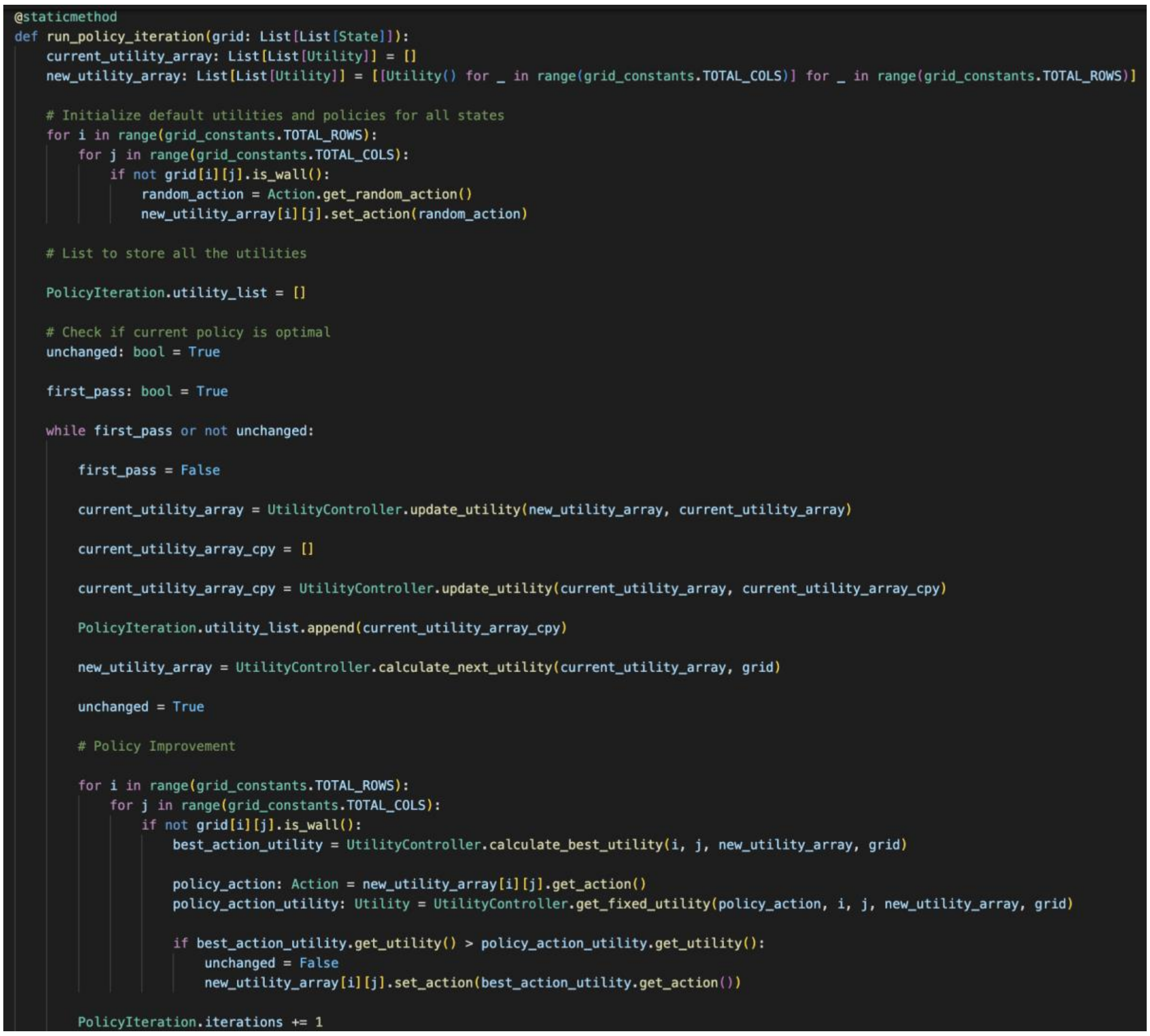
```python
@staticmethod
def run_policy_iteration(grid: List[List[State]]):
    current_utility_array: List[List[Utility]] = []
    new_utility_array: List[List[Utility]] = [[Utility() for _ in range(grid_constants.TOTAL_COLS)] for _ in range(grid_constant

    # Initialize default utilities and policies for all states
    for i in range(grid_constants.TOTAL_ROWS):
        for j in range(grid_constants.TOTAL_COLS):
            if not grid[i][j].is_wall():
                random_action = Action.get_random_action()
                new_utility_array[i][j].set_action(random_action)
```

Fig 28: Initializing vectors and selecting random action

During each iteration, a utility list is generated to store the projected utility values for all states. When policy improvement yields no changes in utilities, a Boolean flag is employed to determine whether the algorithm should cease. The updated utility values are computed using the Bellman equation, leveraging the current utilities and actions of all states. To achieve a sufficiently accurate approximation of utility, 'K' value iteration steps are executed.

Initially, the current utility list is initialized to 0.0, temporarily storing utilities while a simplified "Bellman update" is executed. The current policy actions and utility values are utilized to generate a new utility vector, and its states are initialized accordingly. The current utility array is updated at the onset of each Kth iteration. The new utility for each state is calculated using the utility_controller class method during each Kth iteration based on the action.

Following the estimation of utilities for all states, the utility value for selecting the action maximizing future states is determined. This value is then compared to the projected utility to evaluate the sufficiency of the existing policy. If the utility gained by executing the optimal action surpasses the current predicted utility value, the policy of that specific state is adjusted with a new policy that potentially maximizes its subsequent states. These steps are reiterated until the algorithm identifies the optimal policy.

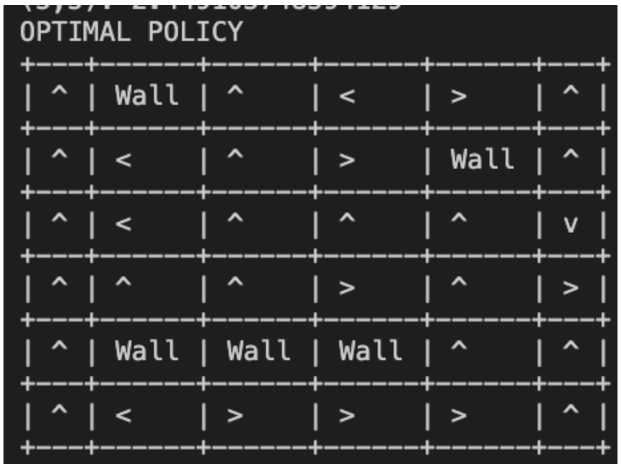


```python
@staticmethod
def run_policy_iteration(grid: List[List[State]]):
    current_utility_array: List[List[Utility]] = []
    new_utility_array: List[List[Utility]] = [[Utility() for _ in range(grid_constants.TOTAL_COLS)] for _ in range(grid_constants.TOTAL_ROWS)]

    # Initialize default utilities and policies for all states
    for i in range(grid_constants.TOTAL_ROWS):
        for j in range(grid_constants.TOTAL_COLS):
            if not grid[i][j].is_wall():
                random_action = Action.get_random_action()
                new_utility_array[i][j].set_action(random_action)

    # List to store all the utilities

    PolicyIteration.utility_list = []

    # Check if current policy is optimal
    unchanged: bool = True

    first_pass: bool = True

    while first_pass or not unchanged:

        first_pass = False

        current_utility_array = UtilityController.update_utility(new_utility_array, current_utility_array)

        current_utility_array_cpy = []

        current_utility_array_cpy = UtilityController.update_utility(current_utility_array, current_utility_array_cpy)

        PolicyIteration.utility_list.append(current_utility_array_cpy)

        new_utility_array = UtilityController.calculate_next_utility(current_utility_array, grid)

        unchanged = True

        # Policy Improvement

        for i in range(grid_constants.TOTAL_ROWS):
            for j in range(grid_constants.TOTAL_COLS):
                if not grid[i][j].is_wall():
                    best_action_utility = UtilityController.calculate_best_utility(i, j, new_utility_array, grid)

                    policy_action: Action = new_utility_array[i][j].get_action()
                    policy_action_utility: Utility = UtilityController.get_fixed_utility(policy_action, i, j, new_utility_array, grid)

                    if best_action_utility.get_utility() > policy_action_utility.get_utility():
                        unchanged = False
                        new_utility_array[i][j].set_action(best_action_utility.get_action())

        PolicyIteration.iterations += 1
```

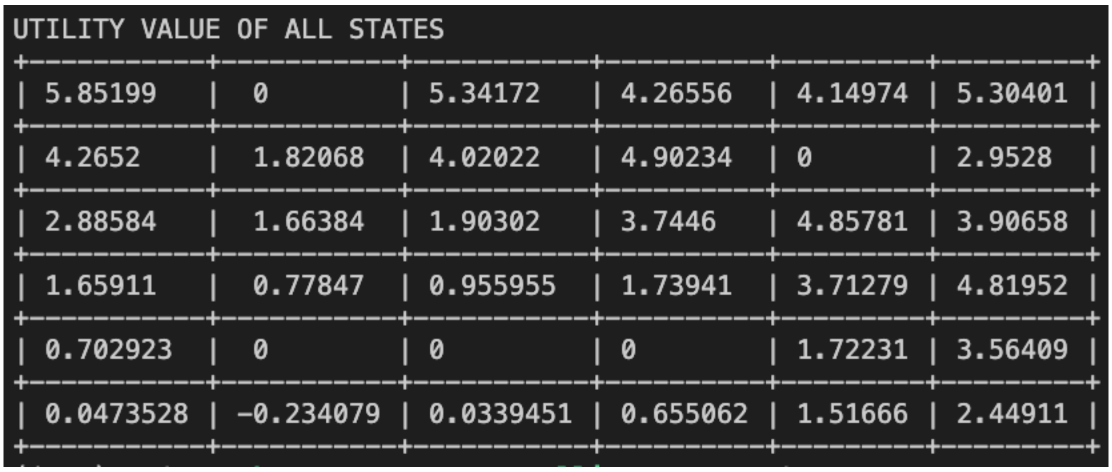
Figure 29: Policy Iteration Implementation

4.2 Optimal Policy relation with “k” value

4.2.1 K = 1

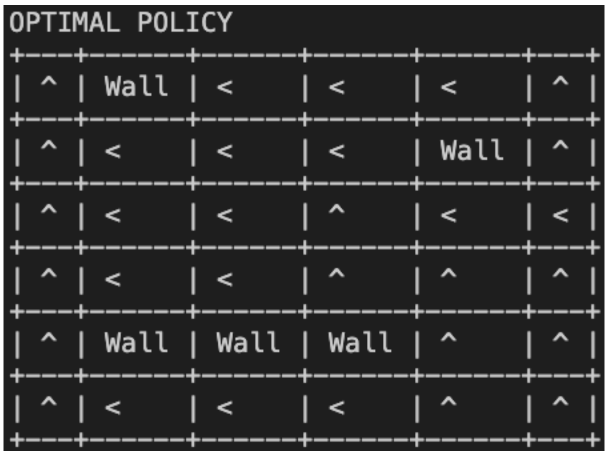


```
OPTIMAL POLICY
+---+------+------+------+------+---+
| ^ | Wall | ^    | <    | >    | ^ |
+---+------+------+------+------+---+
| ^ | <    | ^    | >    | Wall | ^ |
+---+------+------+------+------+---+
| ^ | <    | ^    | ^    | ^    | v |
+---+------+------+------+------+---+
| ^ | ^    | ^    | >    | ^    | > |
+---+------+------+------+------+---+
| ^ | Wall | Wall | Wall | ^    | ^ |
+---+------+------+------+------+---+
| ^ | <    | >    | >    | >    | ^ |
+---+------+------+------+------+---+
```

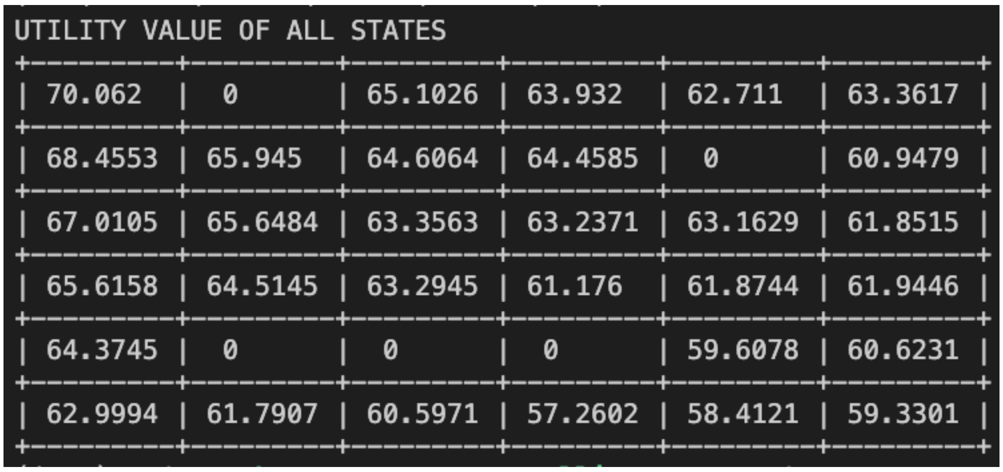


```
UTILITY VALUE OF ALL STATES
+-----------+-----------+-----------+----------+---------+---------+
| 5.85199   |  0        | 5.34172   | 4.26556  | 4.14974 | 5.30401 |
+-----------+-----------+-----------+----------+---------+---------+
| 4.2652    |  1.82068  | 4.02022   | 4.90234  | 0       | 2.9528  |
+-----------+-----------+-----------+----------+---------+---------+
| 2.88584   |  1.66384  | 1.90302   | 3.7446   | 4.85781 | 3.90658 |
+-----------+-----------+-----------+----------+---------+---------+
| 1.65911   |  0.77847  | 0.955955  | 1.73941  | 3.71279 | 4.81952 |
+-----------+-----------+-----------+----------+---------+---------+
| 0.702923  |  0        | 0         | 0        | 1.72231 | 3.56409 |
+-----------+-----------+-----------+----------+---------+---------+
| 0.0473528 | -0.234079 | 0.0339451 | 0.655062 | 1.51666 | 2.44911 |
+-----------+-----------+-----------+----------+---------+---------+
```

4.2.2 K = 20

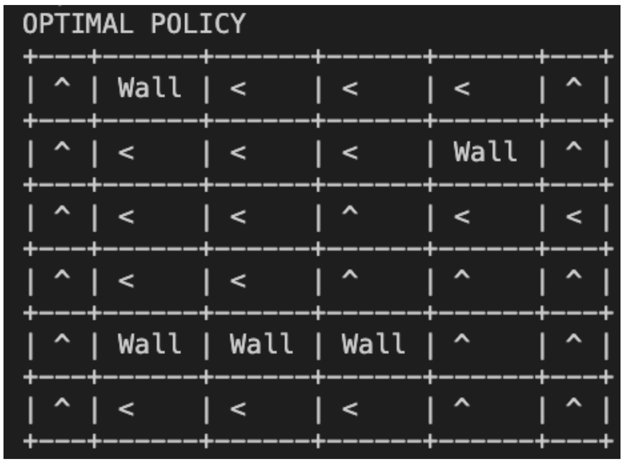


```
OPTIMAL POLICY
+---+------+------+------+------+---+
| ^ | Wall | <    | <    | <    | ^ |
+---+------+------+------+------+---+
| ^ | <    | <    | <    | Wall | ^ |
+---+------+------+------+------+---+
| ^ | <    | <    | ^    | <    | < |
+---+------+------+------+------+---+
| ^ | <    | <    | ^    | ^    | ^ |
+---+------+------+------+------+---+
| ^ | Wall | Wall | Wall | ^    | ^ |
+---+------+------+------+------+---+
| ^ | <    | <    | <    | ^    | ^ |
+---+------+------+------+------+---+
```

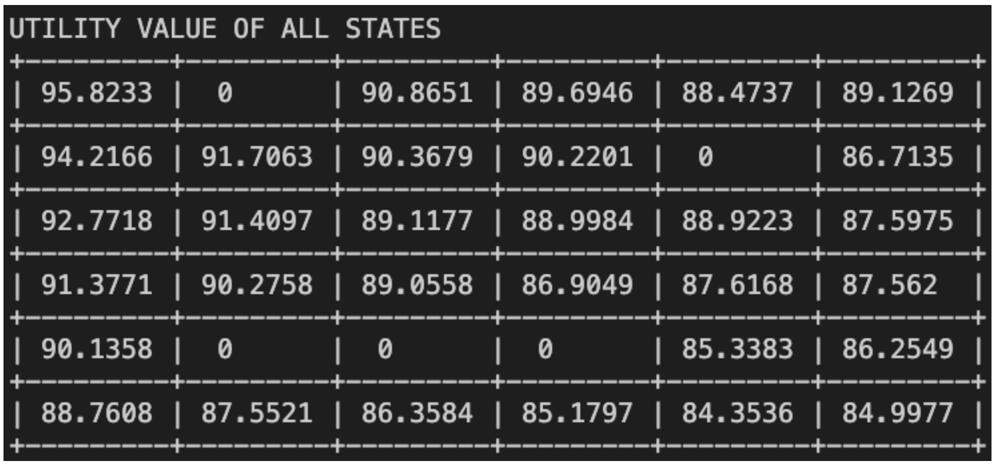


```
UTILITY VALUE OF ALL STATES
+---------+---------+---------+---------+---------+---------+
| 70.062  |  0      | 65.1026 | 63.932  | 62.711  | 63.3617 |
+---------+---------+---------+---------+---------+---------+
| 68.4553 | 65.945  | 64.6064 | 64.4585 |  0      | 60.9479 |
+---------+---------+---------+---------+---------+---------+
| 67.0105 | 65.6484 | 63.3563 | 63.2371 | 63.1629 | 61.8515 |
+---------+---------+---------+---------+---------+---------+
| 65.6158 | 64.5145 | 63.2945 | 61.176  | 61.8744 | 61.9446 |
+---------+---------+---------+---------+---------+---------+
| 64.3745 |  0      |  0      |  0      | 59.6078 | 60.6231 |
+---------+---------+---------+---------+---------+---------+
| 62.9994 | 61.7907 | 60.5971 | 57.2602 | 58.4121 | 59.3301 |
+---------+---------+---------+---------+---------+---------+
```

4.2.3 K = 50



```
OPTIMAL POLICY
+---+------+------+------+------+---+
| ^ | Wall | <    | <    | <    | ^ |
+---+------+------+------+------+---+
| ^ | <    | <    | <    | Wall | ^ |
+---+------+------+------+------+---+
| ^ | <    | <    | ^    | <    | < |
+---+------+------+------+------+---+
| ^ | <    | <    | ^    | ^    | ^ |
+---+------+------+------+------+---+
| ^ | Wall | Wall | Wall | ^    | ^ |
+---+------+------+------+------+---+
| ^ | <    | <    | <    | ^    | ^ |
+---+------+------+------+------+---+
```



```
UTILITY VALUE OF ALL STATES
+---------+---------+---------+---------+---------+---------+
| 95.8233 |  0      | 90.8651 | 89.6946 | 88.4737 | 89.1269 |
+---------+---------+---------+---------+---------+---------+
| 94.2166 | 91.7063 | 90.3679 | 90.2201 |  0      | 86.7135 |
+---------+---------+---------+---------+---------+---------+
| 92.7718 | 91.4097 | 89.1177 | 88.9984 | 88.9223 | 87.5975 |
+---------+---------+---------+---------+---------+---------+
| 91.3771 | 90.2758 | 89.0558 | 86.9049 | 87.6168 | 87.562  |
+---------+---------+---------+---------+---------+---------+
| 90.1358 |  0      |  0      |  0      | 85.3383 | 86.2549 |
+---------+---------+---------+---------+---------+---------+
| 88.7608 | 87.5521 | 86.3584 | 85.1797 | 84.3536 | 84.9977 |
+---------+---------+---------+---------+---------+---------+
```

Figures 30-35: Optimal policies for different K values

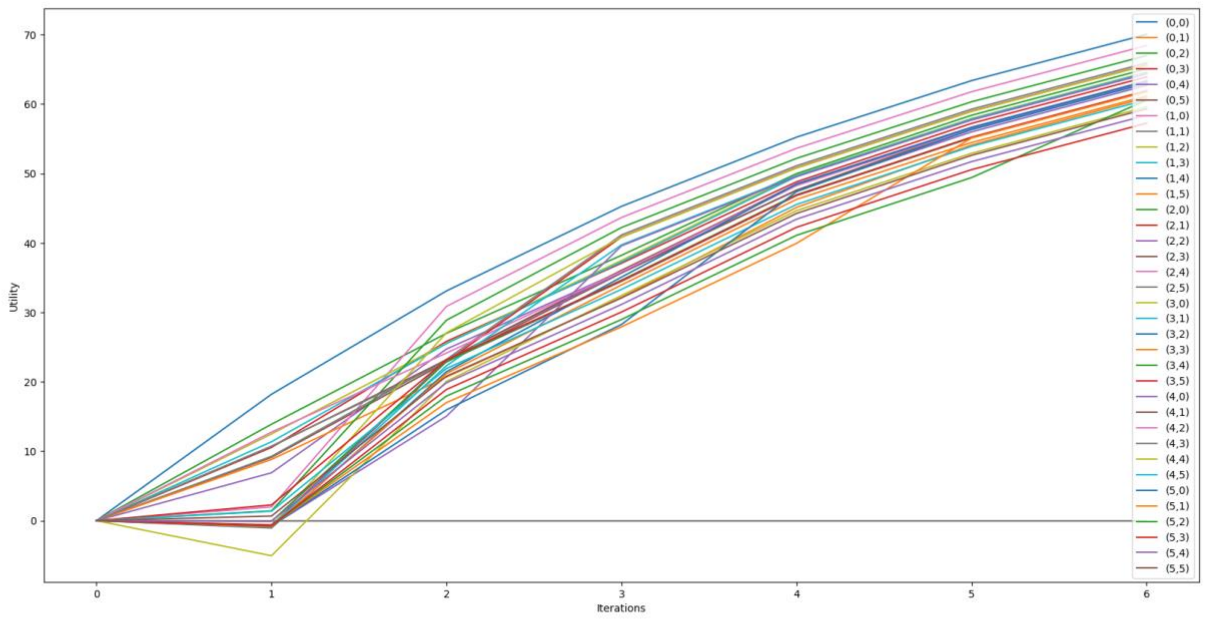


Figure 31: Iteration vs Utility for K = 20

5. Make Maze more Complex

Intuitively, the maze can be made more complex simply by scaling it. Because all utility policies need to be updated at once, increasing the size of the scale has a significant impact on the performance of the two algorithms. This can be experimented based on the same codebase but with more number of cells (updating the constant in the grid_constant.py file). Upon experimenting, it was observed that by simply scaling the grid, the time it took to reach an optimal policy was increasing exponentially. The program crashed after scaling it by 78 times. However, before that, the time it took to calculate the policy was in complete defiance of the quickness with which it worked on a 6x6 grid.