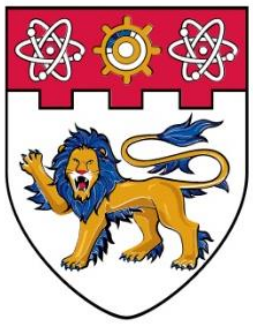


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CZ4046 INTELLIGENT AGENTS

ASSIGNMENT 1 REPORT

Name: **Bhatia Ritik**

Matriculation Number: **U1822529C**

Tutorial Group: **CS4**

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

ACADEMIC YEAR 2021/22

Contents

1. Problem

1.1. Overview

Given a maze environment, write a program to formulate the environment as a Markov Decision Process and solve the same using Value Iteration and Policy Iteration.

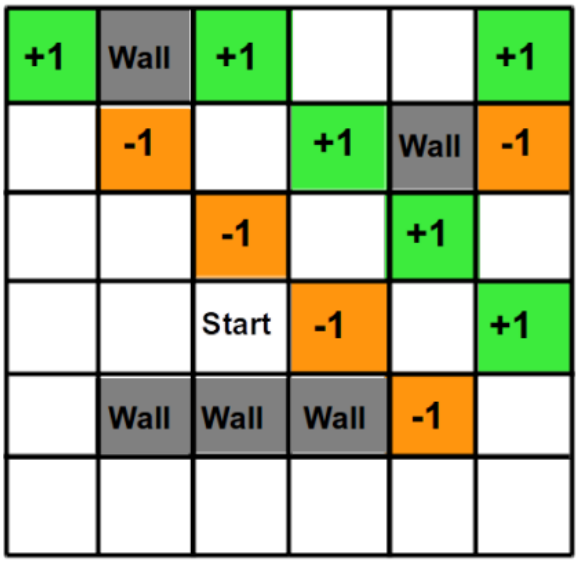


Figure 1. *The maze environment which the agent is in*

The environment of the agent is a grid with 6 rows and 6 columns. Each cell of the grid can be one of the following 4 types:

- Green cell, with a reward of +1
- Brown cell, with a reward of -1
- White cell, with a reward of -0.04
- Grey cell (Wall), with a reward of 0

Depending on the next cell the agent lands in, the above reward function tells the reward that the agent will get for taking the corresponding action.

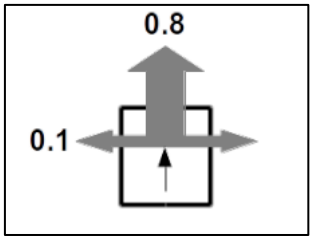


Figure 2. *Transition model for the agent*

Each action taken by the agent is guided by a transition model which is as follows:

- The intended outcome of the action taken by the agent occurs with a probability of 0.8

- With probability 0.1, the agent moves at either right angle to the intended direction.
- If the move would make the agent walk into a wall, the agent stays in the same place as before and no change in position is recorded.

Finally, there are no terminal states, that is, the agent's state sequence is infinite.

1.2. Some observations

From the reference utilities provided in the problem description, the utility of cell (0, 0) is the highest (although with a different discount factor), showing that the optimal policy that we derive using discount factor = 0.99 **should make the agent transition to cell (0, 0)**.

Further, from inspection of the grid, it is clear that at that state, the agent should choose to go UP. This is because if the agent chooses to go UP, it will always end up in the cell (0, 0) and get a reward of +1, irrespective of the direction it eventually ends up going in (due to the nature of the transition model). If the agent does so, it can potentially earn **infinite reward** by continuously doing this single action, and hence the optimality of the state and the corresponding action.

2. Code Overview

2.1. Directory Structure

Logical division of the code is done to ensure proper **readability, accessibility and maintainability**. Each different service has its own class and methods to expose different features, so that sub-problems can be solved by simply calling the methods of the objects.

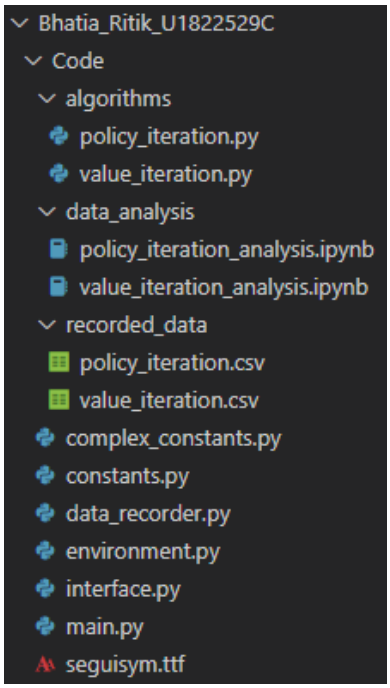


Figure 3. *Directory structure of the code*

From Figure 3, the following deductions can be made:

- The *Code* directory contains the entire code to solve the problem.
- The *algorithms* directory within the *Code* directory contains files to implement the Value Iteration and Policy Iteration algorithms.
- The *data_analysis* directory contains the *Jupyter* notebooks for analysis of the data recorded during the execution of Value Iteration and Policy Iteration algorithms.
- The *recorded_data* directory contains the data recorded for analysis during the execution of the two algorithms.
- Rest of the files are kept under the parent *Code* directory.

The details of each file are as follows:

- *policy_iteration.py*: Implements the Policy Iteration algorithm and provides the methods for the same.

- *value_iteration.py*: Implements the Value Iteration algorithms and provides the methods for the same.
- *policy_iteration_analysis.ipynb*: The *Jupyter* notebook that contains code snippets for analysis of the data recorded during the execution of the Policy Iteration algorithm.
- *value_iteration_analysis.ipynb*: The *Jupyter* notebook that contains code snippets for analysis of the data recorded during the execution of the Value Iteration algorithm.
- *policy_iteration.csv*: CSV file with the data recorded during the execution of the Policy Iteration algorithm.
- *value_iteration.csv*: CSV file with the data recorded during the execution of the Value Iteration algorithm.
- *complex_constants.py*: A file that defines constant, global variables to be used during execution of either algorithm. This file is imported when the **complicated maze environment** is to be used.
- *constants.py*: A file that defines constant, global variables to be used during execution of either algorithm. This file is imported when the maze environment provided in the problem statement is to be used.
- *data_recorder.py*: Provides specification for the class and its methods to convert the dictionary data into a CSV file. The input is expected to be a Python dictionary containing the required data.
- *environment.py*: Specifies the maze environment and formulates it as a Markov Decision Process by providing methods to get the transition model etc.
- *interface.py*: Provides methods to initialize and setup the **PyGame** window (frontend) that displays the optimal policy and the utilities of each cell after the execution of either algorithm.
- *main.py*: **Driver code** to make objects and call relevant methods to execute the Value Iteration and Policy Iteration algorithms.

3. Algorithm Implementation (Part 1)

3.1. Maze Environment

The Maze Environment is simulated as a Markov Decision Process by implementing the *Environment* class in the *environment.py* file. This class contains several helper methods to retrieve the list of actions, reward at each state, transition model etc.

3.1.1. Maze

The maze is defined as a two-dimensional list as shown in Figure 4. Each cell in the grid can be one of the following:

- *"G"*: Corresponds to the green block with a reward of +1
- *"B"*: Corresponds to the brown block with a reward of -1
- *"wall"*: Corresponds to the wall with a reward of 0
- *""*: Corresponds to the white block with a reward of -0.04

```
grid = [
    ['G', 'wall', 'G', '', '', 'G'],
    ['', 'B', '', 'G', 'wall', 'B'],
    ['', '', 'B', '', 'G', ''],
    ['', '', '', 'B', '', 'G'],
    ['', 'wall', 'wall', 'wall', 'B', ''],
    ['', '', '', '', '', ''],
]
```

Figure 4. *Definition of the maze environment*

3.1.2. Rewards

The reward of each type of cell has already been provided as part of the problem statement and hence has been stored as a constant when initializing the Markov Decision Process environment.

The reward for different types of cells are stored as shown in Figure 5. As can be seen, a Python dictionary is used to map the cell content to the corresponding reward (the agent will receive a reward of +1 if it lands in a cell with content "*G*", -1 if it lands in a cell with content "*B*" etc). This is followed by iterating through each cell of the grid, determining the type and storing the corresponding reward in a separate **two-dimensional list *rewards***.

Hence, *rewards[i][j]* is the reward for the cell in row number "*i*" and column number "*j*".

```
reward_mapping = {
    'wall': 0,
    'G': 1,
    'B': -1,
    '': -0.04
}

rewards = [[reward_mapping[grid[row][col]]
            for col in range(grid_width)] for row in range(grid_height)]
```

Figure 5. *Definition of rewards in the code*

Since the agent can never enter a wall, the reward for a wall cell is initialized as 0.

3.1.3. Actions

In any cell within the grid, the agent can choose to go either UP, DOWN, LEFT or RIGHT. The list of actions is stored in an *actions* list in the code, as shown in Figure 6 below.

```
actions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
```

Figure 6. *List of actions that the agent can take*

The *actions* list contains tuples, where each tuple represents the **offset** to be added **to the current coordinate** of the agent for the corresponding action. The mapping of actions is:

- (-1, 0): UP
- (1, 0): DOWN
- (0, -1): LEFT
- (0, 1): RIGHT

For example, if the current coordinate of the agent within the grid is (x, y) and the agent decides to go UP, this means that the agent will remain in the same column, however, the row number will decrease by 1, making the final position as (x – 1, y). We get the same result when we add the specified offset of (-1, 0) to (x, y) as (x, y) + (-1, 0) = (x – 1, y).
A similar justification can be provided for the rest of the actions.

When adding the offset to the current coordinate, appropriate checks have been placed in the code to ensure that the new position of the agent is valid, that is, **within the bounds of the grid and is not on a wall cell**. This can be clearly seen in the code snippet in Figure 7.

```
        for probability, direction in dir_and_probability:
            new_row = row + direction[0]
            new_col = col + direction[1]

            if 0 <= new_row < self.get_grid_height() and 0 <= new_col < self.get_grid_width():
                if self.grid[new_row][new_col] == "wall":
                # rest of the code below this line has been omitted for brevity
```

Figure 7. *Sanity checking for update coordinates after adding offset*

For displaying the policy on the PyGame interface, each action is mapped to the arrow depicting the direction of movement of the agent. The code snippet that implements the mapping is shown in Figure 8 below.

```
# mapping for action to arrow symbol
ACTION_TUPLE_CONVERSION = {(1, 0): '↓', (-1, 0): '↑',
                           (0, 1): '→', (0, -1): '←'}
```

Figure 8. *Mapping of the agent to the arrow direction*

3.1.4. Transition Model

The transition model has been specified by the problem statement, which states that the intended outcome of the action taken by the agent occurs with a probability of 0.8, while the agent moves in either direction at a right angle to the intended direction with a probability of 0.1. If the direction of movement of the agent makes it go into a wall, it stays in the same place. The transition model has been implemented as shown in Figure 9.

The transition model gives ***P(s' | s, action)*** where *s'* is the new state and *s* is the current state, that is, given the current state and the action that the agent takes, the transition model returns the **probability of the agent ending up in the new state *s'***.
Hence, to simulate the same, the ***get_transition_model()*** function is created that accepts 3 parameters – the current row and column (which together specify the current state) and the intended action of the agent. The actual implementation involves considering all possible directions of movement for the said action (e.g., for action UP, the agent may move UP, LEFT or RIGHT), getting the new coordinates of the agent and recording the **total probability** of the agent ending up in the updated coordinates.

The *transition_model* is a dictionary that maps the coordinates to the probability of the agent ending up in those coordinates.

```python
def get_transition_model(self, row, col, action):
    '''
    Definition
    ----------

    Returns the transition model P(s'|s,a) for a particular state and action


    Parameters
    ----------

    row : int
        The row (indexed from 0) of the specified state

    col : int
        The column (indexed from 0) of the specified state

    action: tuple
        The action that is being taken

    '''

    # initialize the transition model as a dictionary to store the mappings
    transition_model = defaultdict(int)

    # dir_and_probability lists the probability of a particular direction of movement
    # as well as the offset to be added to the current coordinates to retrieve the
    # updated coordinates
    dir_and_probability = [
        [0.8, action],
        [0.1, (-action[1], -action[0])],
        [0.1, (action[1], action[0])]
    ]

    # iterate over all the possible directions of movement
    for probability, direction in dir_and_probability:
        new_row = row + direction[0]
        new_col = col + direction[1]

        # process further only if the new coordinates are valid coordinates
        if 0 <= new_row < self.get_grid_height() and 0 <= new_col < self.get_grid_width():

            # if the new coordinates are that of a wall, then the agent stays in the current state
            if self.grid[new_row][new_col] == "wall":
                new_row, new_col = row, col

        # otherwise the agent remains in the current state
        else:
            new_row, new_col = row, col

        # add the probability to the updated state entry in the transition model
        transition_model[(new_row, new_col)] += probability

    # return the transition model to the caller
    return transition_model
```

Figure 9. *Code snippet implementing the transition model of the agent*

In the above code snippet, we **add *probability* to the entry in the *transition_model*** because the agent may end up in the **same state** as before, if it encounters a wall, thereby increasing the probability of staying in the same state.

Finally, the transition model is returned to the caller for further use.

3.2. Value Iteration

The basic idea of Value Iteration is to calculate the utility of each state and then use the state utilities to select an optimal action in each state.

As per the reference book, the utility of a state is defined as the **immediate reward for that state plus the expected discounted utility of the next state**, assuming that the agent chooses an optimal action. The formula of the utility of a state is given in Figure 10 below.

$$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) U(s')$$

Figure 10. *Formula for the utility of a state*

The equation shown in Figure 10 is called the **Bellman Equation** and this equation forms the crux of the Value Iteration algorithm. The equation presents the utility of any state as a **function of its neighbors** and is continually performed until the utility of every state converges.

The pseudocode of the Value Iteration algorithm, as mentioned in the reference book - *"Artificial Intelligence: A Modern Approach" by S. Russell and P. Norvig. Prentice-Hall, third edition, 2010*, is shown in Figure 11 below.

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 11. *Pseudocode of the Value Iteration algorithm*

The pseudocode shown in Figure 11 has been used for the code implementation of the Value Iteration algorithm, the details of which will be covered in the following sections.

3.2.1. Defining constants

The constants for the Value Iteration algorithm are shown in Figure 12 below.

```
# value iteration settings
val_iter_discount_factor = 0.99
val_iter_scaler = 0.05
MAX_REWARD = 1
val_iter_error = val_iter_scaler * MAX_REWARD
```

Figure 12. *Constants for Value Iteration algorithm*

Based on Figure 12 above, the constants defined for the Value Iteration algorithm are:

- Discount factor = 0.99
- Maximum Reward = +1
- Error scaler = 0.05
- Error = Error scaler * Maximum Reward

3.2.2. Solving the Markov Decision Process

The function ***solve_mdp()*** to solve the Markov Decision Process takes 2 arguments: *mdp* and the *error*, as specified in the pseudocode.

The important steps in the code and the corresponding code snippets are:

1. Initialize the utility of each cell in the grid to 0.

```
# initialize the utility of each cell as 0 before the value iteration
utilities = [[0 for i in range(mdp.get_grid_width())]
             for j in range(mdp.get_grid_height())]
```

Figure 13. *Initializing the utility of each cell as 0*

2. Set exit condition ***threshold*** with the formula provided in the pseudocode in Fig 11.

```
# calculate change threshold for terminating value iteration loop
threshold = error * (1 - self.discount_factor) / self.discount_factor
```

Figure 14. *Threshold for exit condition*

3. Initialize the analysis data for each cell with a 0.

```
# initilize analysis data for each cell
for row in range(mdp.get_grid_height()):
    for col in range(mdp.get_grid_width()):
        analysis_data_key = "(" + str(col) + "," + str(row) + ")"
        self.analysis_data[analysis_data_key] = [0]
```

Figure 15. *Initialization of the analysis data*

4. Perform the following steps until the exit condition is met:
   a) Iterate through each cell in the grid.
   b) If the cell is not a wall, iterate through the possible actions the agent can take.
   c) Given the current cell and action, **retrieve the transition model** and get the **utility of taking the current action**.

      The utility of taking the current action can be found by multiplying the probability of the direction of movement and the transition utility (the utility of the new state after moving in the specified direction).

```
# iterate through each possible action
for action in mdp.get_actions():
    curr_action_utility = 0

    # retrieve the transition model: P(s'|s, a)
    transition_model = mdp.get_transition_model(
        row, col, action)

    # iterate through each new state in the transition model
    # the robot can be in a state different than the intended one
    for new_state in transition_model:
        transition_probability = transition_model[new_state]
        transition_utility = utilities[new_state[0]
                                       ][new_state[1]]

        # add the utility of taking current action
        curr_action_utility += transition_probability * transition_utility
```

Figure 16. *Calculation of utility after taking current action*

   d) Record current utility if it is greater than utilities of any of the previous actions.
   e) Update the utility of the current cell by **discounting** the optimal action's utility and adding the reward of entering the new cell.

```
# collect reward for transition
reward = mdp.get_reward(row, col)

# record the updated utility
updated_utilities[row][col] = reward + self.discount_factor * optimal_action_utility
```

Figure 17. *Updating utility of current cell*

f) Record the maximum change in utility.

```
# record change in utility as a result of the step
max_utility_change = max(
    max_utility_change, abs(updated_utilities[row][col] - utilities[row][col]))
```

Figure 18. *Recording maximum change in utility so far*

g) Store the updated utilities.

h) If the **maximum change in utility is lesser than the threshold for change**, then the exit condition is met, and the loop can be exited.

```
# if the change in utility across all cells is smaller than the change threshold, exit
if max_utility_change < threshold:
    break
```

Figure 19. *Exit condition for loop*

5. Find the optimal policy after the utilities of all cells have converged.
6. Return the required information to the caller.

As can be seen from the above steps, they closely resemble the ones suggested in the pseudocode shown in Figure 11. The algorithm runs until the utility of each cell has converged by performing Bellman updates and finds the optimal policy after this.
The steps for finding the optimal policy are detailed in section 3.2.3. below.

3.2.3. Finding the optimal policy

The function ***get_optimal_policy()*** finds the optimal policy using the utility value of each cell. It accepts 2 parameters: the *mdp* and the *utilities* list calculated earlier while solving the MDP.

To find the optimal policy, this function takes the following steps:

1. The policy is initialized to go DOWN for each cell in the grid.

```
# initialize the policy to go down at all cells
policy = [[(1, 0) for col in range(mdp.get_grid_width())]
          for row in range(mdp.get_grid_height())]
```

Figure 20. *Policy initialization*

2. Iterate over each cell in the grid and do the following:

a) Initialize the optimal action to *None* and its utility to *-ve infinity*.

b) For each possible action that the agent can take, get the transition model, and calculate the utility of executing that action.

```
# iterate over all the actions
for curr_action in mdp.get_actions():

    # initiallize utility of current action to 0
    curr_action_utility = 0

    # retrieve the transition model: P(s'|s,a)
    transition_model = mdp.get_transition_model(
        row, col, curr_action)

    # iterate through each new state in the transition model
    # the robot can be in a state different than the intended one due to the nature of the transition model
    for new_state in transition_model:
        transition_probability = transition_model[new_state]
        transition_utility = utilities[new_state[0]
                                       ][new_state[1]]

        # add the utility of transition multiplied by the probability of that transition happening
        curr_action_utility += transition_probability * transition_utility
```

Figure 21. *Finding utility of all possible actions*

c) Record the **action with the maximum utility** and store it as the optimal action to be taken for the specified cell.

```
# if utility of current action is more than the best one so far
# then the current action is the optimal action so far
if curr_action_utility > optimal_action_utility:
    optimal_action = curr_action
    optimal_action_utility = curr_action_utility
```

Figure 22. *Recording most optimal action*

3. Return the policy to the caller.

Finding the optimal policy after calculating cell utilities is fairly straightforward. Following the above steps gets the optimal action to take at every cell, which is the optimal policy itself.

3.2.4. Executing the Value Iteration algorithm

To execute the Value Iteration algorithm, the following steps are taken in the *main.py* file:

1. The environment of the Markov Decision Process, the interface for the PyGame display and an object of the *DataRecorder* class are initialized.

```
# initialize variables used for both value iteration and policy iteration
interface = Interface(cell_size, height, width)
mdp = Environment(grid, grid_height, grid_width, actions, rewards)
data_recorder = DataRecorder("recorded_data/")
```

Figure 23. *Variable initialization*

2. An object of class *ValueIteration* is created with the given discount factor.
3. The *solve_mdp()* function of the above object is invoked to solve the Markov Decision Process (MDP). The parameters to this function are the MDP itself and the **maximum allowable error** in the utility value of each cell.

```
value_iteration = ValueIteration(val_iter_discount_factor)
result = value_iteration.solve_mdp(mdp, val_iter_error)
```

Figure 24. *Method invocation to solve the MDP*

4. Once solved, the results are displayed in the terminal and the PyGame interface.

```
print("Number of iterations = " + str(num_iters))
print()
print("Cell-wise utilities: (Col, Row)")
print()
# display the utilities of each cell, formatted as (col, row), in the command line
for row in range(grid_height):
    for col in range(grid_width):
        print("(" + str(col) + "," + str(row) + "): " +
              str(utilities[row][col]))
print()

# display policy on the pygame interface
direction_array = [[ACTION_TUPLE_CONVERSION[optimal_policy[row][col]]
                    for col in range(grid_width)] for row in range(grid_height)]
interface.display(arr=direction_array, grid=grid, offset=POLICY_CELL_OFFSET,
                  font=POLICY_FONT, title='Value Iteration Policy')

# display utilities on the pygame interface
utility_values = [["{:.2f}".format(utilities[row][col]) for col in range(
    grid_width)] for row in range(grid_height)]
interface.display(arr=utility_values, grid=grid, offset=UTILITY_CELL_OFFSET,
                  font=UTILITY_FONT, title='Value Iteration Utilities')
```

Figure 25. *Display of results*

5. Finally, the analysis data is stored in a CSV file for later analysis.

```
# record data of algorithm execution into a csv, for future data analysis
data_recorder.record("value_iteration.csv",
                     value_iteration.get_analysis_data())
```

Figure 26. *Storing analysis data in a CSV file*

3.2.5. Results

With the **error scaling factor set to 0.05**, the Value Iteration algorithm takes **757 iterations** until the utility value of each cell converges, post which it proceeds to find the optimal policy.

Figure 27 shows the final utility values of each cell (rounded to **2 decimal places**) after the successful completion of the Value Iteration algorithm.

99.95		95.00	93.83	92.60	93.28
98.34	95.83	94.50	94.35		90.87
96.90	95.54	93.24	93.13	93.05	91.75
95.50	94.40	93.18	91.07	91.76	91.84
94.26				89.50	90.52
92.89	91.68	90.49	89.31	88.52	89.25

Figure 27.*Utility of each cell after Value Iteration*

In Figure 28, we can see the optimal policy found by the algorithm. The direction of the arrows in each cell represents the optimal direction of movement for the agent.

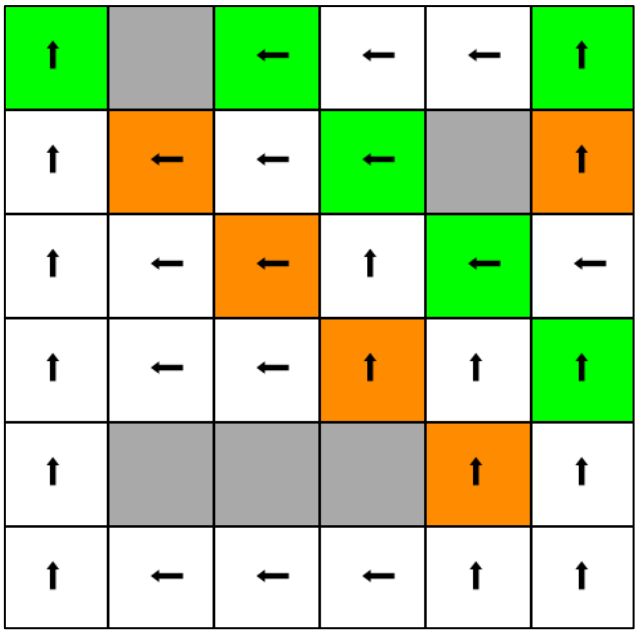
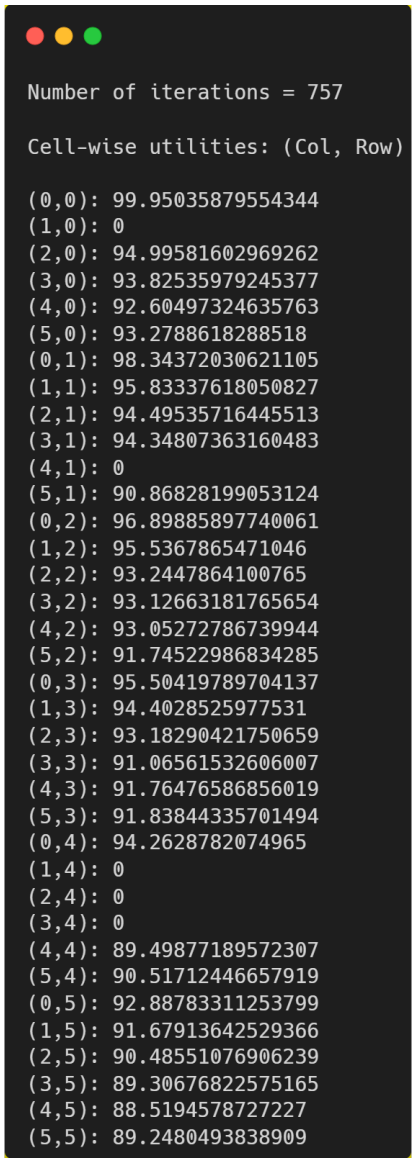


Figure 28. *Optimal policy based on final utilities*

Figure 29 shows the terminal output of the Value Iteration algorithm, depicting the total number of iterations and the utility of each cell in the grid. The coordinates of a cell in the grid (in the output below) is in the format ***(column, row)***.



```
Number of iterations = 757

Cell-wise utilities: (Col, Row)

(0,0): 99.95035879554344
(1,0): 0
(2,0): 94.99581602969262
(3,0): 93.82535979245377
(4,0): 92.60497324635763
(5,0): 93.2788618288518
(0,1): 98.34372030621105
(1,1): 95.83337618050827
(2,1): 94.49535716445513
(3,1): 94.34807363160483
(4,1): 0
(5,1): 90.86828199053124
(0,2): 96.89885897740061
(1,2): 95.5367865471046
(2,2): 93.2447864100765
(3,2): 93.12663181765654
(4,2): 93.05272786739944
(5,2): 91.74522986834285
(0,3): 95.50419789704137
(1,3): 94.4028525977531
(2,3): 93.18290421750659
(3,3): 91.06561532606007
(4,3): 91.76476586856019
(5,3): 91.83844335701494
(0,4): 94.2628782074965
(1,4): 0
(2,4): 0
(3,4): 0
(4,4): 89.49877189572307
(5,4): 90.51712446657919
(0,5): 92.88783311253799
(1,5): 91.67913642529366
(2,5): 90.48551076906239
(3,5): 89.30676822575165
(4,5): 88.5194578727227
(5,5): 89.2480493838909
```

Figure 29. *Terminal output - Number of iterations and utility of each cell*

Finally, the graph in Figure 30 is a plot of the **utility estimates of each cell against the number of iterations**. As can be seen from the graph, for the first 200 iterations, the utility values of each cell change **significantly** and make big steps towards the final value. In the final few iterations, the utility values of each cell **converge very slowly** to the final value with minimal change in utilities across iterations. This signifies that the Value Iteration algorithm is close to finding the optimal values and may terminate in the next few iterations.

Please note: the number of iterations will change a lot as the value of the *error scaler* changes. **Greater the value of the *error scaler*** (that is, greater the allowable error in the utility value of

each cell), **fewer the number of iterations**. However, this also implies a greater error in the utility values of each cell. After a certain threshold, if the *error scaler* is increased any further, the Value Iteration algorithm would not be able to find the optimal policy due to large errors in the utility values. Hence there exists an **upper bound** for the value of the *error scaler*.

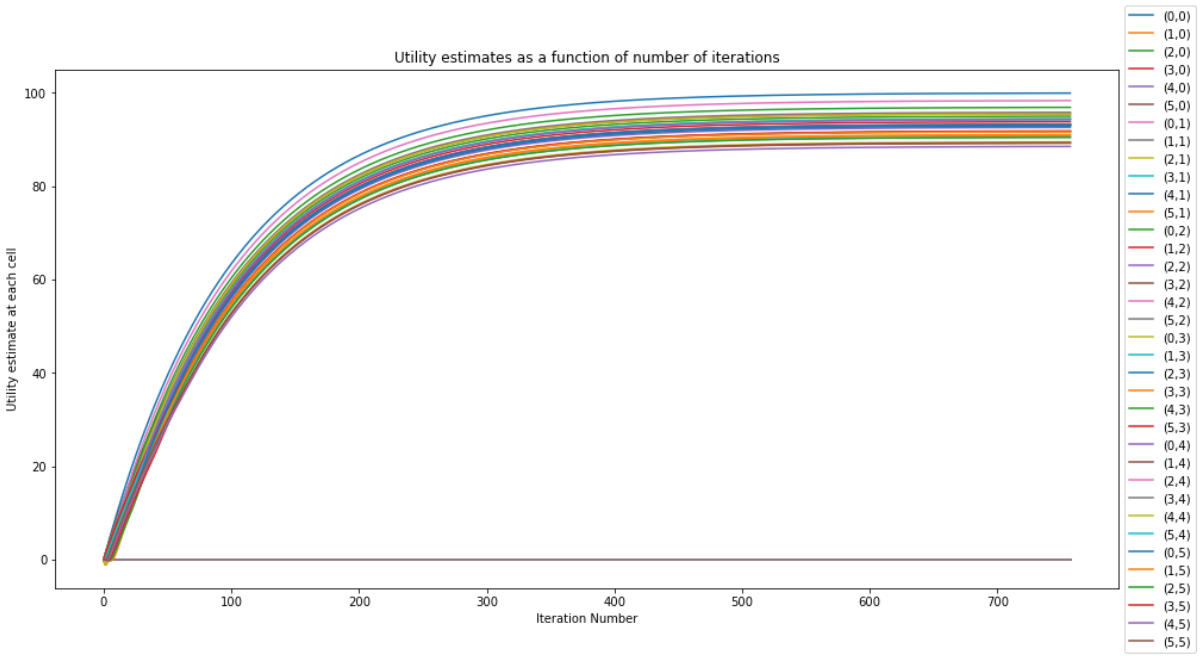


Figure 30. *Plot of utility estimates vs iterations for Value Iteration*

From the above results, it is easy to infer that the Value Iteration algorithm guarantees to find an optimal policy for the provided Markov Decision Process.

3.3. Policy Iteration

Policy Iteration is another algorithm to find the optimal policy for a Markov Decision Process by alternating between 2 main steps:

- Policy Evaluation – given a policy, the utility of each state is calculated if that policy were to be executed.
- Policy Improvement – a new policy with the **maximum expected utility** is calculated using one-step lookahead.

As can be seen from the pseudocode shown in Figure 31 below (from the reference book - *"Artificial Intelligence: A Modern Approach" by S. Russell and P. Norvig. Prentice-Hall, third edition, 2010*) the important steps in the Policy Iteration Algorithm are:

1. **Policy Evaluation is the first step** where the utilities of each cell are calculated by following the current policy (the initial policy can be something random, for e.g. go RIGHT in all cells).

2. Bellman update is applied ***k* times** (a parameter of the algorithm) to calculate the utility of each cell. The formula of Bellman update is shown below:

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

3. The next step is **Policy Improvement** where the current policy is updated by using the new values of utilities calculated in Step 2 above. The action is chosen per the formula below, that is, the action that maximizes the expected utility.

$$\pi[s] \leftarrow \underset{a \in A(s)}{\operatorname{argmax}} \sum_{s'} P(s' \mid s, a)\ U[s']$$

4. Steps 1-3 are repeated **until there is no change in the policy improvement step**, that is, when the policy has **stabilized**.

function POLICY-ITERATION(mdp) **returns** a policy
  inputs: mdp, an MDP with states S, actions $A(s)$, transition model $P(s' \mid s, a)$
  local variables: U, a vector of utilities for states in S, initially zero
    π, a policy vector indexed by state, initially random

  repeat
    $U \leftarrow$ POLICY-EVALUATION(π, U, mdp)
    $unchanged? \leftarrow$ true
    for each state s **in** S **do**
      if $\max_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\ U[s'] > \sum_{s'} P(s' \mid s, \pi[s])\ U[s']$ **then do**
        $\pi[s] \leftarrow \underset{a \in A(s)}{\operatorname{argmax}} \sum_{s'} P(s' \mid s, a)\ U[s']$
        $unchanged? \leftarrow$ false
  until $unchanged?$
  return π

Figure 31. *Pseudocode for Policy Iteration algorithm*

A key takeaway from the Policy Iteration algorithm is that there is no need to calculate the exact / precise utilities of each state. We can use simplifications that can give **reasonably good approximation of utilities** and that is usually sufficient to get the optimal policies.

The above is quite useful for large state spaces where solving *n* equations in *n* unknowns becomes exponentially difficult.

The details of the code implementation will be covered in the following sections.

3.3.1. Defining constants

The constants for the Policy Iteration algorithm are shown in Figure 32 below.

```
# policy iteration settings
policy_iter_discount_factor = 0.99
policy_iter_num_policy_eval_iters = 100
```

Figure 32. *Constants for Policy Iteration algorithm*

Based on Figure 32 above, the constants defined for the Policy Iteration algorithm are:

- Discount factor = 0.99
- Number of steps in Policy Evaluation (k) = 100

3.3.2. Solving the Markov Decision Process

The function ***solve_mdp()*** to solve the Markov Decision Process takes 1 argument: *mdp*, as specified in the pseudocode.

The important steps in the code and the corresponding code snippets are:

1. Set the initial policy – the default action to be taken at each cell, since the algorithm has not yet started. In our case, we have initialized it to go RIGHT at each cell.

```
def get_initial_policy(self, mdp):
    random_start_policy = [[(0, 1) for col in range(
        mdp.get_grid_width())] for row in range(mdp.get_grid_height())]
    return random_start_policy
```

Figure 33. *Policy Initialization*

2. Initialize the utility and analysis data entry for each cell with a 0.

```
# initialize the utility of each cell as 0 before the value iteration
utilities = [[0 for col in range(mdp.get_grid_width())]
             for row in range(mdp.get_grid_height())]

# initilize analysis data for each cell
for row in range(mdp.get_grid_height()):
    for col in range(mdp.get_grid_width()):
        analysis_data_key = "(" + str(col) + "," + str(row) + ")"
        self.analysis_data[analysis_data_key] = [0]
```

Figure 34. *Utility and analysis_data initalization*

3. Iterate while the policy is still **unstable**:
    a. Evaluate the policy, that is, find the utility of each cell given the current policy.

b. Improve the policy, that is, find the updated optimal action at each cell given the updated utility values.

```
# iterate while the policy is still changing at each improvement step
while not policy_unchanged:

    # evaluate policy to get utilies at each cell
    utilities = self.evaluate_policy(mdp, utilities, policy)
    num_iters += self.num_policy_eval_iters

    # improve policy based on the updated utilities, to get the final optimal policy
    policy, policy_unchanged = self.improve_policy(mdp, utilities, policy)
```

Figure 35. *Iteration steps until the policy is stable*

4. Return the required values to the caller.

```
# Return the required information to the caller
return {
    "num_iters": num_iters,
    "utilities": utilities,
    "optimal_policy": policy
}
```

Figure 36*. Information returned at the end*

3.3.3. Policy Evaluation

The ***evaluate_policy()*** method calculates the utility of each state given the current policy. It accepts 3 arguments – *mdp*, *utilities* and *policy*.

The important steps in its implementation are as follows:

1. Iterate for each cell in the grid:
   a. If the current cell is not a wall, retrieve the transition model *P(s'| s,a)* where *a* is the current action given by the current policy.

```
# initialize the current action and set its utility to 0
curr_action = policy[row][col]
curr_action_utility = 0

# retrieve the transition model: P(s'|s,a)
transition_model = mdp.get_transition_model(
    row, col, curr_action)
```

Figure 37. *Retrieval of the transition model*

   b. For each new state that the agent can go to, add the expected transition utility to the utility of the current action being taken. This is done by multiplying the transition utility to the probability of the transition happening.

```
# iterate through each new state in the transition model
# the robot can be in a state different than the intended one due to the nature of the transition model
for new_state in transition_model:
    transition_probability = transition_model[new_state]
    transition_utility = utilities[new_state[0]
                                   ][new_state[1]]

    # add the utility of transition multiplied by the probability of that transition happening
    curr_action_utility += transition_probability * transition_utility
```

Figure 38. *Calculation of the utility of the current action*

c. Get the updated utility for the cell by adding the reward with the discounted utility of taking the current action.

```
# collect reward for transition
reward = mdp.get_reward(row, col)

# record the updated utility
updated_utilities[row][col] = reward + self.discount_factor * curr_action_utility
```

Figure 39. *Retrieval of updated utilities*

d. Update the analysis data.

2. Carry out the above step for ***k* steps**, where *k* is number of iterations during evaluation.
3. Return the utilities to the caller.

3.3.4. Policy Improvement

The algorithm improves upon the previous policy by finding **new optimal actions in each cell** based on the updated utility values. This is done by the ***improve_policy()*** method that takes 3 arguments – *mdp*, *utilities* and *policy*.

The important steps in its implementation are:

1. For each cell in the grid, iterate over all the possible actions that the agent can take.
2. Retrieve the transition model for the given state and action and calculate the utility of the current action by iterating over the possible new states that the agent can go to.

```
# iterate through each new state in the transition model
# the robot can be in a state different than the intended one due to the nature of the transition model
for new_state in transition_model:
    transition_probability = transition_model[new_state]
    transition_utility = utilities[new_state[0]
                                   ][new_state[1]]

    # add the utility of transition multiplied by the probability of that transition happening
    curr_action_utility += transition_probability * transition_utility
```

Figure 40. *Calculation of the utility of the current action*

3. If the utility of the current action is the maximum so far, mark it as the optimal action.

```
# if utility of current action is more than the best one so far
# then the current action is the optimal action so far
if curr_action_utility > optimal_action_utility:
    optimal_action = curr_action
    optimal_action_utility = curr_action_utility
```

Figure 41. *Recording most optimal action for the current cell*

4. Record the optimal action for each cell and return the improved policy to the caller.

3.3.5. Executing the Policy Iteration algorithm

To execute the Policy Iteration algorithm, the following steps are taken in the *main.py* file:

1. The environment of the Markov Decision Process, the interface for the PyGame display and an object of the *DataRecorder* class are initialized.

```
# initialize variables used for both value iteration and policy iteration
interface = Interface(cell_size, height, width)
mdp = Environment(grid, grid_height, grid_width, actions, rewards)
data_recorder = DataRecorder("recorded_data/")
```

Figure 42. *Variable initialization*

2. An object of class *PolicyIteration* is created with the given discount factor.
3. The *solve_mdp()* function of the above object is invoked to solve the MDP. The parameters to this function are the MDP itself and the maximum allowable error in the utility value of each cell.

```
# if the user has chosen the run policy iteration
elif choice == 2:
    policy_iteration = PolicyIteration(
        policy_iter_discount_factor, policy_iter_num_policy_eval_iters)
    result = policy_iteration.solve_mdp(mdp)
```

Figure 43. *Method invocation to solve the MDP*

4. Once solved, the results are displayed in the terminal and the PyGame interface.

```
print("Number of iterations = " + str(num_iters))
print()
print("Cell-wise utilities: (Col, Row)")
print()
for row in range(grid_height):
    for col in range(grid_width):
        print("(" + str(col) + "," + str(row) + "): " +
              str(utilities[row][col]))
```

Figure 44. *Display of results*

5. Finally, the analysis data is stored in a CSV file for later analysis.

3.3.6. Results

With **100 iterations for the Policy Evaluation step**, the Policy Iteration algorithm takes **700 iterations until convergence**.

Figure 45 shows the final utility values of each cell (rounded to 2 decimal places) after the successful completion of the Policy Iteration algorithm. As we can see, the utility values differ slightly from that after the Value Iteration algorithm but are nevertheless close to those values.

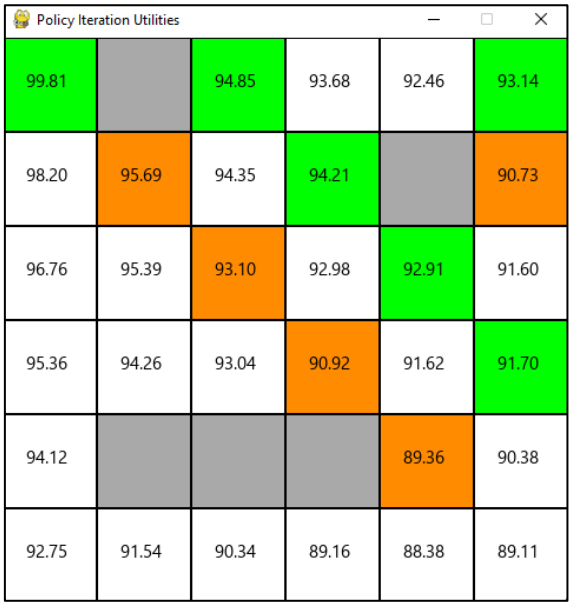


Figure 45.*Utility of each cell after Policy Iteration*

In Figure 46, we can see the optimal policy found by the algorithm. The direction of the arrows represents the direction of movement of the agent in that cell.

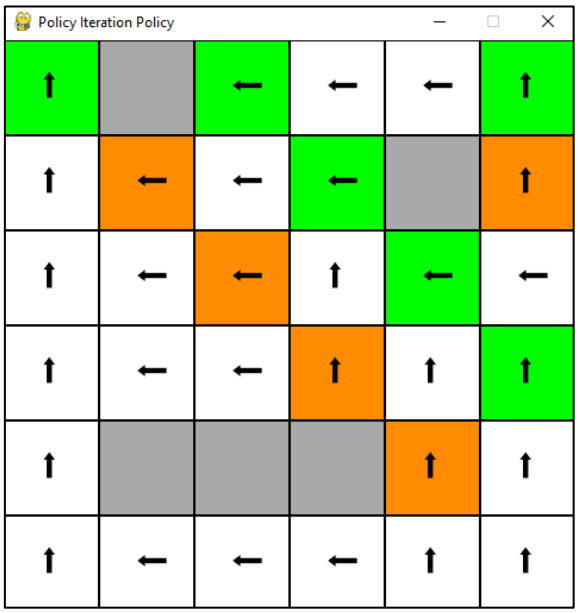
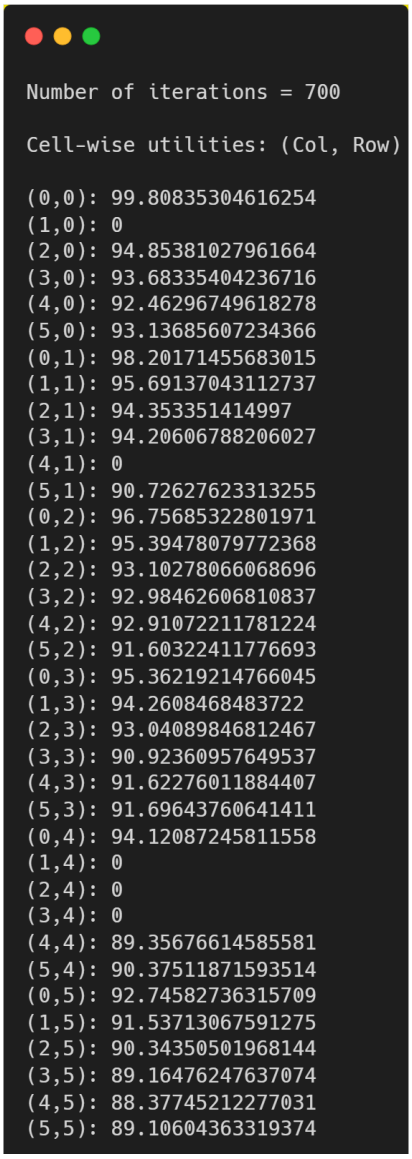


Figure 46. *Optimal policy*

Figure 47 shows the terminal output of the Policy Iteration algorithm, depicting the total number of iterations and the utility of each cell in the grid. The coordinates of a cell in the grid (in the output below) is in the format ***(column, row)***.



```
Number of iterations = 700

Cell-wise utilities: (Col, Row)

(0,0): 99.80835304616254
(1,0): 0
(2,0): 94.85381027961664
(3,0): 93.68335404236716
(4,0): 92.46296749618278
(5,0): 93.13685607234366
(0,1): 98.20171455683015
(1,1): 95.69137043112737
(2,1): 94.353351414997
(3,1): 94.20606788206027
(4,1): 0
(5,1): 90.72627623313255
(0,2): 96.75685322801971
(1,2): 95.39478079772368
(2,2): 93.10278066068696
(3,2): 92.98462606810837
(4,2): 92.91072211781224
(5,2): 91.60322411776693
(0,3): 95.36219214766045
(1,3): 94.2608468483722
(2,3): 93.04089846812467
(3,3): 90.92360957649537
(4,3): 91.62276011884407
(5,3): 91.69643760641411
(0,4): 94.12087245811558
(1,4): 0
(2,4): 0
(3,4): 0
(4,4): 89.35676614585581
(5,4): 90.37511871593514
(0,5): 92.74582736315709
(1,5): 91.53713067591275
(2,5): 90.34350501968144
(3,5): 89.16476247637074
(4,5): 88.37745212277031
(5,5): 89.10604363319374
```

Figure 47. *Terminal output - Number of iterations and utility of each cell*

Finally, the graph in Figure 48 is a plot of the utility estimates of each cell against the number of iterations. As can be seen from the graph, certain iterations are marked by a **"steep jump"** in the value of utility (occurring at iteration number 100, 200 etc). This is because there are 100 iterations in each Policy Evaluation step. The jumps can also be attributed to the **algorithm finding the optimal action** that maximized the expected utility after policy improvement.

Another observation is that the increase in utilities is significant near the lower iterations than the higher iterations. This is because as the number of iterations increase, the algorithm is that

much closer to convergence, and hence the minimal change in utility values of each cell. That can also signify that the policy has become **relatively stable**.

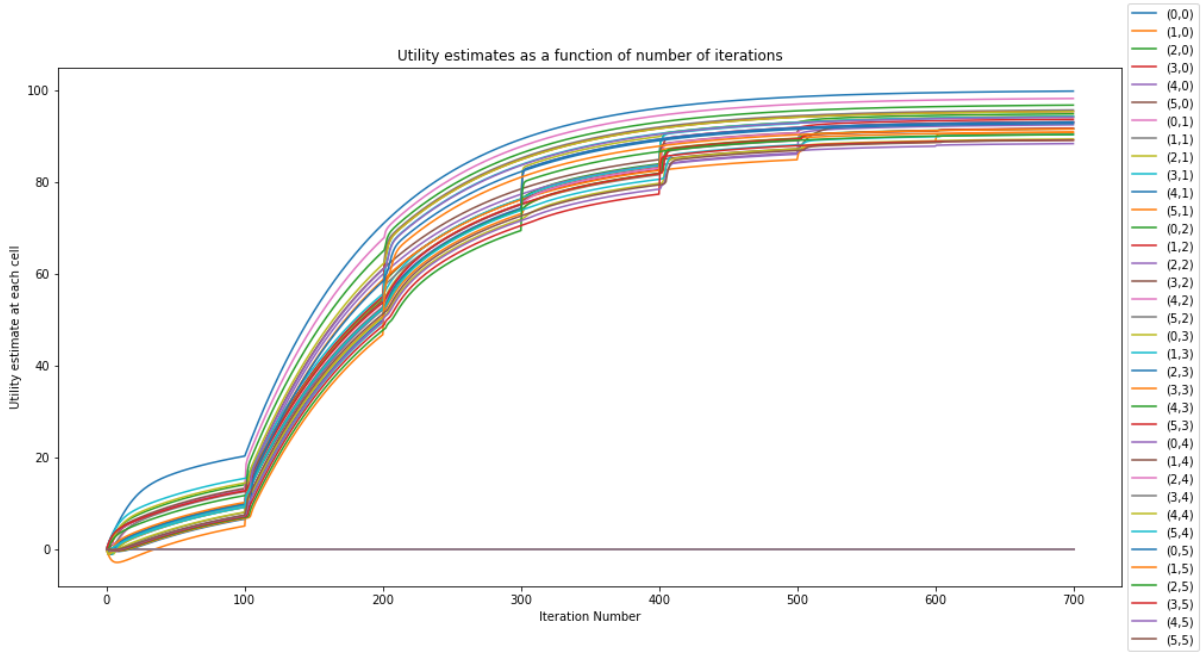


Figure 48. *Plot of utility estimates vs iterations for Policy Iteration*

From the above results, it is easy to infer that the Policy Iteration algorithm also guarantees to find an optimal solution to the provided Markov Decision Process. This is because, as expected, the algorithm suggests that the optimal action for the agent to perform in the cell (0, 0) is to go up, by doing which, the agent can receive **infinite reward** (this is also justified by the high utility value of the cell).

4. Complicated Maze Environment (Part 2)

A more complicated maze environment was designed to study the effects of the number of states and the complexity of the maze on the convergence of both Value Iteration and Policy Iteration.

To run the complicated version of the maze environment, the file ***complex_constants.py*** has to be imported to the main file, as it contains the driver code to implement a more complicated version of the maze.

The code to implement the complicated maze is shown in Figure 49 below.

```
# grid settings
np.random.seed(42)
grid_width = 18
grid_height = 18
grid = []

for row in range(grid_height):
    grid_row = []
    for col in range(grid_height):
        random_num = uniform(0, 1)
        if random_num <= 0.25:
            grid_row.append("wall")
        elif random_num <= 0.4:
            grid_row.append("G")
        elif random_num <= 0.55:
            grid_row.append("B")
        else:
            grid_row.append("")
    grid.append(grid_row)
```

Figure 49. *Implementation of the complicated maze environment*

As can be seen from the code in Figure 59 above, the complicated maze environment is a grid of size **18x18**. The maze is constructed by iterating through each cell and for each cell, a **random number** between 0 and 1 is generated (by using Python's ***random*** library). Following is the mapping of the value to the type of cell:

- If 0 < *random_num* <= 0.25: The cell is of type *wall*
- If 0.25 < *random_num* <= 0.4: The cell is of type *G* (reward = +1)
- If 0.4 < *random_num* <= 0.55: The cell is of type *B* (reward = -1)
- If 0.55 < *random_num* <= 1: The cell is of type *empty* (reward = -0.04)

From the above mapping, it is clear that 25% of the cells will of type *wall*, 15% of the cells will be of type *G* (green), 15% of the cells will be of type *B* (brown) and 40% of the cells will be of type *empty* (white).

4.1. Value Iteration results on complicated maze

Figure 50 below shows the optimal policy found by executing the Value Iteration algorithm on a complicated maze environment, with a greater number of states than the Markov Decision Process environment originally solved.

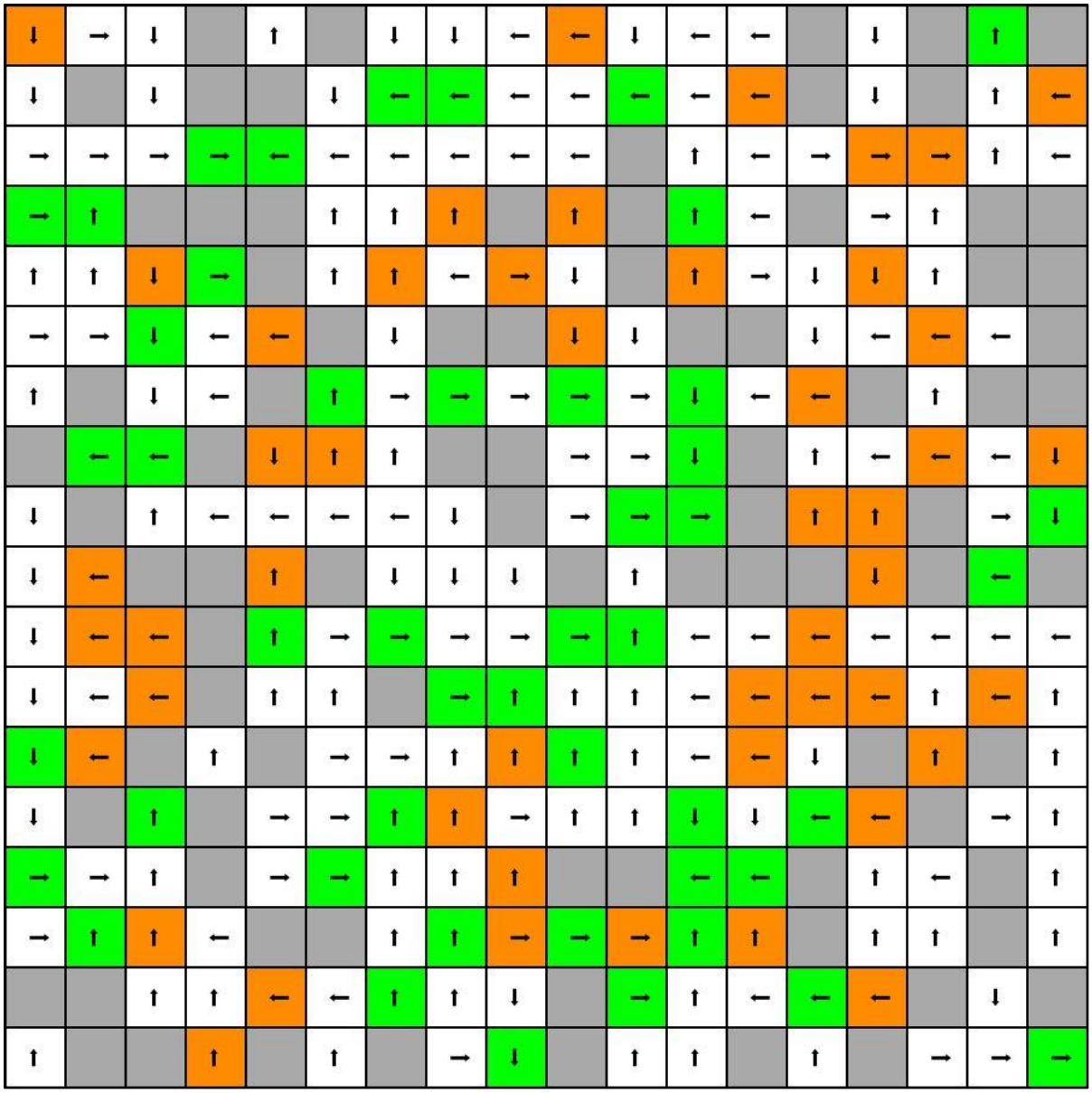


Figure 50. *Optimal policy by Value Iteration on complicated maze*

A general observation that can be made from the above output is that at every cell, the Value Iteration algorithm outputs the correct decision to try to remain in a green square or take an action that will help the agent reach a green square, for greater reward.

Similarly, the cell utilities after executing the Value Iteration algorithm on the complicated maze environment is given in Figure 51 below.

92.46	94.51	95.75		-4.00		95.55	95.22	94.11	91.89	92.10	91.02	89.81		88.96		99.90	
94.63		97.17			96.97	96.85	96.58	95.10	93.57	93.48	91.98	89.59		90.14		98.29	95.77
95.88	97.17	98.45	99.90	99.90	98.27	96.82	95.44	94.30	92.86		90.64	89.47	90.14	91.33	93.92	96.54	95.34
96.92	97.17				96.82	95.46	93.27		90.45		90.61	89.48		91.12	92.52		
95.68	95.85	95.54	96.89		95.30	93.16	92.09	90.24	92.64		88.41	89.85	91.08	89.28	91.09		
95.12	96.45	97.80	96.59	94.14		92.51			94.17	95.35			92.67	91.22	88.95	87.79	
93.88		98.09	96.79		94.29	93.73	95.27	95.22	96.47	96.76	98.15	96.87	94.07		87.79		
	99.90	99.54		92.46	91.99	92.45			96.13	97.25	98.46		92.66	91.18	88.68	87.54	86.51
89.78		98.09	96.82	94.95	93.53	92.37	91.83		96.94	98.32	98.59		90.11	88.94		87.61	88.74
90.97	88.76			92.52		92.82	92.97	93.98		97.04				87.30		87.93	
92.44	90.16	88.01		92.62	92.65	94.02	94.09	95.34	96.77	96.89	95.48	93.99	91.38	89.67	88.34	87.04	85.77
93.95	92.49	89.86		91.41	91.44		95.13	95.38	95.49	95.48	94.26	91.97	89.91	87.78	87.02	84.95	84.68
95.37	92.88		-4.00		91.25	92.45	93.69	93.24	95.19	94.23	93.20	91.07	91.19		84.69		83.57
95.62		99.90		89.93	91.11	92.30	91.48	92.31	93.71	93.14	93.71	92.47	92.41	89.85		81.27	82.35
96.88	97.14	98.44		89.94	91.13	91.02	90.33	89.96			93.86	93.54		88.52	87.24		81.27
95.78	96.91	95.92	94.50			89.88	90.23	88.87	91.20	91.09	93.40	91.38		87.24	86.22		80.20
		94.50	93.14	90.73	89.41	89.85	89.06	88.27		91.88	92.03	90.89	90.86	88.34		97.17	
-4.00			90.73		88.24		88.27	89.33		90.70	90.82		89.67		97.17	98.45	99.90

Figure 51. *Cell utilities after Value Iteration on complicated maze*

The Value Iteration algorithm on the complicated maze environment took **688 iterations** to find the optimal policy and converge on the utility values of all the cells.

From the above figures, it is clear that the Value Iteration algorithm is **still able to find the optimal policy** for the complicated maze environment, even though it takes slightly longer to execute. Although at certain cells the optimal policy can vary (as the agent might have an option to choose one of many possible optimal actions), the above policy provided by the algorithm is **still optimal**.

An informal proof of the same is that in cells where the agent can choose an action to get infinite reward (by always ending up in the same cell), the utility values of those cells are nearly 99.9, which suggests that the algorithm was able to find the optimal policy.

4.2. Policy Iteration results on complicated maze

Figure 52 below shows the optimal policy found by executing the Policy Iteration algorithm on a complicated maze environment, with a greater number of states than the Markov Decision Process environment originally solved.

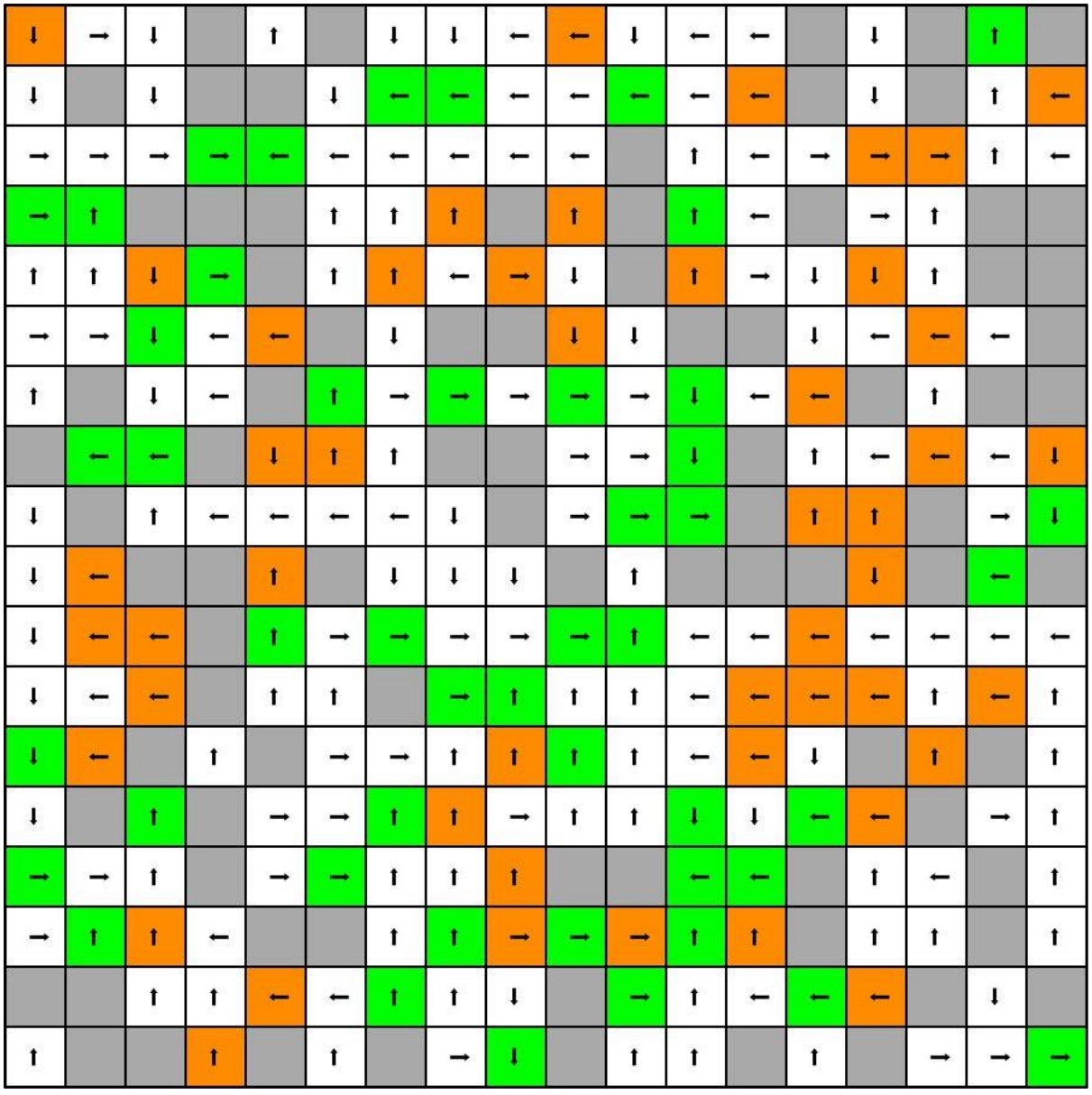


Figure 52. *Optimal policy after Policy Iteration on complicated maze*

Similar to the output of Value Iteration, an observation that can be made here is that at every cell, the Policy Iteration algorithm outputs the correct decision to try to remain in a green square or take an action that will help the agent reach a green square, for greater reward.

Similarly, the cell utilities after executing the Policy Iteration algorithm on the complicated maze environment is given in Figure 53 below.

92.55	94.60	95.84		-4.00		95.64	95.32	94.20	91.99	92.20	91.12	89.90		89.06		100.00	
94.73		97.26			97.07	96.94	96.67	95.19	93.66	93.57	92.07	89.69		90.23		98.39	95.86
95.97	97.26	98.54	99.99	99.99	98.36	96.92	95.53	94.39	92.95		90.73	89.56	90.23	91.42	94.02	96.64	95.44
97.02	97.27				96.92	95.56	93.37		90.55		90.71	89.57		91.21	92.62		
95.78	95.94	95.63	96.99		95.39	93.25	92.18	90.34	92.74		88.51	89.95	91.18	89.38	91.19		
95.21	96.55	97.89	96.69	94.23		92.60			94.26	95.45			92.76	91.32	89.05	87.89	
93.98		98.19	96.89		94.39	93.82	95.37	95.31	96.57	96.85	98.24	96.97	94.16		87.89		
	100.00	99.64		92.56	92.08	92.55			96.22	97.34	98.55		92.76	91.27	88.78	87.63	86.61
89.88		98.19	96.92	95.05	93.62	92.47	91.93		97.03	98.41	98.69		90.21	89.03		87.71	88.83
91.06	88.85			92.62		92.91	93.07	94.08		97.14				87.40		88.02	
92.54	90.25	88.11		92.71	92.74	94.12	94.19	95.44	96.86	96.98	95.57	94.08	91.48	89.76	88.43	87.13	85.86
94.04	92.58	89.95		91.51	91.53		95.23	95.48	95.59	95.57	94.35	92.07	90.01	87.87	87.12	85.04	84.77
95.47	92.98		-4.00		91.35	92.55	93.79	93.34	95.29	94.32	93.29	91.17	91.29		84.78		83.67
95.72		100.00		90.02	91.21	92.39	91.57	92.41	93.81	93.24	93.81	92.57	92.51	89.94		81.36	82.44
96.98	97.23	98.54		90.04	91.23	91.12	90.42	90.06			93.96	93.64		88.61	87.33		81.36
95.88	97.01	96.01	94.60			89.98	90.33	88.97	91.30	91.19	93.50	91.47		87.33	86.32		80.30
		94.60	93.24	90.83	89.50	89.95	89.15	88.36		91.98	92.13	90.99	90.95	88.44		97.27	
-4.00			90.83		88.34		88.36	89.43		90.80	90.91		89.77		97.27	98.54	100.00

Figure 53. *Cell utilities after Policy Iteration on complicated maze*

The Policy Iteration algorithm on the complicated maze environment took **1200 iterations** until the policy stabilized.

Similar to Value Iteration, an informal proof of the correctness of the Policy Iteration algorithm is that for the cells in the grid above where the agent can choose an action to get infinite reward (by always ending up in the same cell), the utility values of those cells are nearly 100.00, which suggests that the algorithm was able to find the optimal policy.

4.3. Effect of number of states and complexity of environment on convergence

On increasing the complexity of the maze environment and the number of states, both algorithms took **significantly longer execution time** than the originally provided maze environment. This shows that as the complexity and the number of states increases, the time taken by the algorithms to converge increases.

Further, the code of both algorithms involves usage of several data structures, for e.g. a list of *utilities* and a dictionary of *transition_model* to track every cell / state in the grid. Thus, the **memory / space usage** of both Value Iteration and Policy Iteration increases.

However, as can be seen from the results of executing both algorithms on the complicated maze environment, both are **guaranteed to find an optimal policy** even if the complexity of the maze and the number of states are increased.

Finally, the reward function used in both algorithms are **discounted**, that is, present rewards have greater weight than future rewards. The equation of the same is shown below.

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

From the above equation, it is clear that as long as the **discount factor (γ) is less than 1, the utility value will be bounded** (which is the case in our implementation of Value Iteration and Policy Iteration as we keep the value of γ = 0.99).
However, if the value of **γ = 1, then the value of the utility will be infinite**, which means that both algorithms will never converge as the utility values will never converge. Both will continue to execute to gain more reward from other states, till infinity.

Hence, to summarise, following are our observations regarding the effect of complexity of maze and number of states on convergence of Value Iteration and Policy Iteration:

- As complexity and number of states increases, time taken for convergence increases.
- As complexity and number of states increases, the memory / space used increases.
- As complexity and number of states increases, the correctness of the 2 algorithms doesn't change, that is, they still find the optimal solution.
- Both the algorithms are guaranteed to converge regardless of the change in complexity and number of states, as long as the discount factor < 1.

4.4. How complex can an environment be and still learn the right policy

Irrespective of the complexity of the maze, both algorithms are **guaranteed to converge** and learn the right policy, as can be proved by the correct results in the complicated maze environment above. As long as the **environment is finite** and the **rewards are discounted** (with discount factor < 1), both Value Iteration and Policy Iteration are guaranteed to converge and return the optimal policy.

Hence, to summarise, if the environment is finite and the rewards are discounted, the **complexity of the maze can be increased without sacrificing the correctness and convergence of both algorithms**. However, if the number of states is infinite or the discount factor >= 1, then there is no guarantee for either Value Iteration or Policy Iteration to converge.

5. Appendix

5.1. Running the Code

To execute the provided code, please follow the below steps:

1. Navigate to the *Code* directory
2. Install the required dependencies by running *pip install -r requirements.txt*
3. Once the dependencies are installed, the program can be executed by running the command *python main.py* in the *Code* directory itself
4. After executing Step 3, the user can see a menu in the terminal window. Choose the option from the menu by entering the option number followed by pressing *Enter*.
5. The policy and utility values will be displayed in a separate *PyGame* window.

To use the complicated maze environment, open the file *main.py* and import the file *complex_constants.py*, instead of *constants.py* (which is imported by default).