CZ4046 Intelligent Agents
Assignment 1 – Agent Decision Making

Name: Glendon Thaiw Yong Neng
Matriculation Number: U1821713F

Contents

1. Understanding the Problem

1.1. Introduction of Grid World Problem

Grid World is a 2D rectangular grid of size Nx, Ny (in our case, Nx=6 and Ny=6 grid) with an agent starting off at one grid square. The agent will take actions in the form of directional steps (up, down, left, or right) to move to an adjacent grid square within the environment of the rectangular grid.

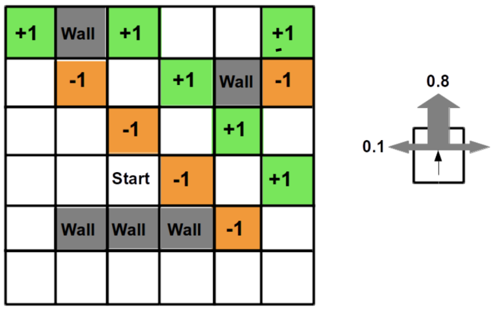


Figure 1: Agent Environment

Referring to the diagram above, there are 4 types of grid squares:

1) Green square: Landing on this square results in the agent receiving a positive 1 reward
2) Brown Square: Landing on this square results in the agent receiving a negative 1 reward
3) White Square: Landing on this square has no effect on the agent in terms of reward and
4) Gray Square/Wall: This square serves as an obstacle - the agent cannot move into this grid square.

1.2. Transition Model

The Transition model has been outlined in the assignment brief as follows: The intended outcome occurs with probability 0.8. The agent moves at either right angle to the intended direction with a probability of 0.1. If the move would make the agent walk into a wall, the agent stays in the same place as before. The rewards for the white squares are -0.04, green squares are +1, brown squares are -1. There are no terminal states; the agent's state sequence is infinite.

1.3. Defining the Objective

The agent environment defined in this assignment illustrates a Markov Decision Process (MDP), where the transition properties depend only on the current state, and not on the previous history or states. Through this assignment, we will implement a program to set up the above environment with an aim to identify the optimal policy for the MDP using 2 methods - 1) value iteration and 2) policy iteration.

2. Formal set up of MDP Components

2.1. States

With our environment of a 6x6 rectangular grid, we have a total of 36 states identified through a cartesian coordinate system of (0,0) to (5,5). In our implementation, the notation of (row,col) has been used to denote states in our environment.

Definition of the coordinate system can be illustrated with the figure below - with (0,0) being at the top left of the agent environment, (0,5) at the top right, (5,0) at the bottom left and (5,5) at the bottom right of the grid environment.

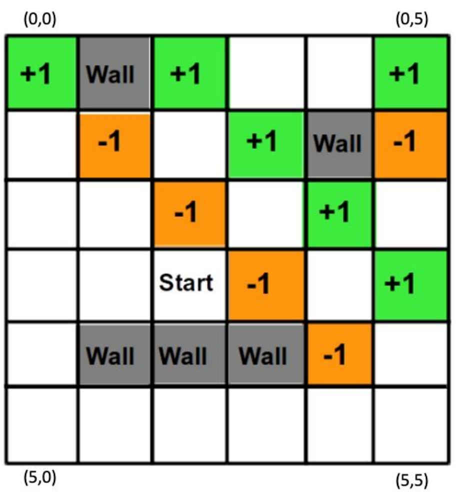


Figure 2: Coordinate System of Agent Environment

2.2. Action A(s)

Each state, s, has actions A(s) available to take at each state. In our case, the maximum number of available actions A(s) on a state are up, down, left, and right. At each state, the agent can move in its intended direction or at right angle to the intended direction. If the move makes the agent walk into a wall, the agent will stay in the same place as before.

2.3. Transition Model P(s' | s, a)

The transition model P(s' | s,a) can be explained by "What's the probability of the next state s', given the current state, s and the action taken, a, in the current state". With the transition model defined in the assignment: the intended outcome occurs with probability 0.8, and 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.

P(s' | s,a) = P(expected s' | s,a) = 0.8
P(s' | s,a) = P(right angle to expected s' | s,a) = 0.1

2.4. Reward Function R(s)

The reward function R(s) describes the amount of reward received by the agent at each state. Formalising the reward system described above, we have the following reward functions of the different types of squares:

R(Green square) = +1
R(Brown square) = -1
R(White square) = -0.04

2.5. Policy $\pi(s)$

The policy π(s) gives us an action that the agent will take in any given state - which is also known as the "solution" to a MDP. The optimal policy should maximise the expected utility over all possible state sequences produced by that policy. We will utilise the Bellman Equation, which represents the recursive relationship between the utilities of successive states. In this definition, the utility of the current state equates to the summation of the reward of the current state and the discounted utility (discount factor of 0.99) of all future rewards. We will attempt to solve this Bellman Equation with two methods: 1) Value iteration and 2) Policy iteration.

3. Abstract of Implemented Solutions

3.1. Source Code Structure

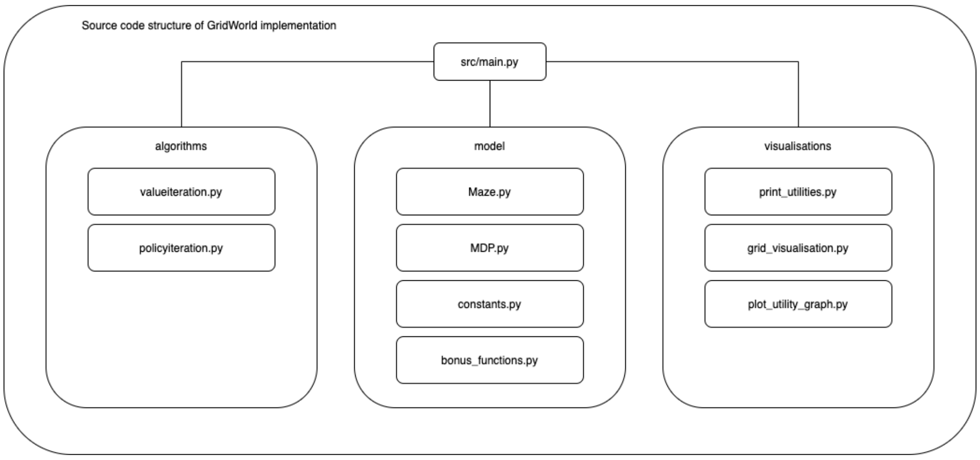


Figure 3: File Structure of Source Code

The source code files used in this program have been thematically organised within the folders - algorithms, model and visualisations based on their respective functionality. With reference to the above hierarchy of the source code structure, the following table describes the functionality of the individual python files.

All Source Files		
File Name	Folder	Functionality
main.py	Source	Entry point of program. This python file is used to execute the different functions implemented in the program to initialise environments, execute value/policy iteration algorithms and generate visualisations.
Maze.py	Model	Represents the maze environment with respective conditions such as action probability values, in each state as shown in the environment.
MPD.py	Model	A abstract class to represent the Markov Decision Process including transition model, reward function and policies.
constants.py	Model	Responsible for the storage of all constant variables used in the program such as Discount Factor, Epsilon value, Reward Values, etc.
bonus_functions.py	Model	Responsible for the generation of more complicated mazes and functions to extract time and number of iterations taken for convergence with varying maze configurations.
valueiteration.py	Algorithm	Responsible for the implementation of the value iteration algorithm.
policyiteration.py	Algorithm	Responsible for the implementation of the policy iteration algorithm.
print_utilities.py	Visualisation	Responsible for printing the final utility of all states to visualise the states of a policy.
grid_visualisation.py	Visualisation	Responsible for the generation of grid Graphical User Interfaces (GUIs) using the Tkinter python library to visualise the optimal policy and final utility values of all states in our grid environment.
plot_utility_graph.py	Visualisation	Responsible for the generation of graphs to visualise the trend of utility estimates as a function of number of iterations.

Table 1: Functionality of Source Code

3.2. Running the Program

The program can be accessed by executing the python file main.py as attached below.

Listing 1: Code snippet of main.py

```
from model.constants import *
from model.Maze import Maze
from model.bonus_functions import generate_maze
from model.bonus_functions import vary_mazesize
from model.bonus_functions import vary_complexity
```

```python

from algorithms.policyiteration import *
from algorithms.valueiteration import *

from visualisations.plot_utility_graph import plot_utility_graph
from visualisations.print_utilities import print_utilities
from visualisations.grid_visualisation import grid_visualisation

import time
import numpy

if __name__ == '__main__':
    ######################################################################
    # 1. Value Iteration Algorithm
    ######################################################################
    # Initialise a maze environment
    maze1 = Maze(GRID, REWARD_MAP, DISCOUNT)

    # Run Value Iteration Algorithm
    value_iteration_res = value_iteration(maze1)

    # Extract outputs from Value Iteration Algorithm
    final_state_utilities = value_iteration_res['U_current']
    optimal_policy = value_iteration_res['optimal_policy']
    U_iterations = value_iteration_res['U_iterations']
    num_iterations = value_iteration_res['num_iterations']

    # Visualise Outputs
    grid_visualisation(optimal_policy, maze1.grid, "policy")
    grid_visualisation(final_state_utilities,maze1.grid,"utilities")
    print_utilities(final_state_utilities, num_iterations,
        algorithm="valueiteration")
    plot_utility_graph(num_iterations, U_iterations, title="Value Iteration")
    ...
    ...
```

The relevant functions for maze creation, executing the optimal policy algorithms and generating visualisations from the algorithms are imported and invoked from this central location.

3.3. Value Iteration

3.3.1. Theory of Value Iteration Algorithm

The value iteration algorithm is used to calculate an optimal policy with a basis of calculating the utility of each state, and then using the state utilities to select an optimal action in each state. It is used to solve the Bellman equation (a system of nonlinear equations) through an iterative approach; We start with an arbitrary initial values for the utilities (in our case, we initialise the utility values of all states to be 0), calculate the right-hand side of the equation, and plug it into the left-hand side—thereby updating the utility of each state from the utilities of its neighbors. We repeat this until we reach an equilibrium.

Let $Ui(s)$ be the utility value for state s at the ith iteration. The iteration bellman update step looks like this:

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

Figure 4: Bellman Update Step

The Bellman update process will be iterated through values of i, with the goal of maximising the absolute utility of all states. This process will continue until the increase in utility for each iteration falls smaller than a declared convergence threshold value (see below) - which we use as a proxy/indication for convergence.

3.3.2. Convergence Threshold Value

The convergence threshold value is used as a mechanism to indicate convergence of the value iteration process which is used to terminate the value-iteration algorithm.

The threshold value is defined by the formula $\frac{e(1-y)}{y}$ where Epilson, e is defined by $e = C * Rmax$.

With the above equations, e represents the maximum error allowed in the utility of the state, C is an arbitrary constant and $Rmax$ is the maximum reward value. Depending on different discount factors and Epsilon values, the results of the optimal policy derived from the value iteration algorithm can vary.

3.3.3. Implementation of Value Iteration Algorithm

Listing 2: Python implementation of Value Iteration Algorithm

```python
def value_iteration(mdp, convergence_threshold):
 U_current, U_next = {}, {}
    max_utility_change =

    # Additional data structures for visualisation
    U_iterations, optimal_policy= {}, {}

    for state in mdp.states:
        #initialise utility value of all states in U_current and U_next to be
            0
        U_current[state] = 0
        U_next[state] = 0
        U_iterations[state] = []
        optimal_policy[state] = None

    num_iterations = 0
    converged = False

    while not converged:
        # U     U    ;        0
        for state in mdp.states:
            U_iterations[state].append(U_current[state])
            U_current[state] = U_next[state] # U    U

        max_utility_change = 0 #

        for state in mdp.states:
            updated_utility, best_action = bellman_update(mdp, state, U_current)
            U_next[state] = updated_utility
            optimal_policy[state] = best_action

            abs_utility_change = abs(U_next[state] - U_current[state])
            if abs_utility_change > max_utility_change:
                max_utility_change = abs_utility_change

        num_iterations += 1

        # print("Iteration: " + str(num_iterations))
        # print("Maximum Utility Change: " + str(max_utility_change))

        if max_utility_change < CONVERGENCE_THRESHOLD:
            converged = True

    return {
        ’U_current’: U_current,
        ’U_iterations’: U_iterations,
        ’optimal_policy’: optimal_policy,
        ’num_iterations’: num_iterations
    }
```

The above code snippet describes the python implementation for the value iteration algorithm. The function takes in 2 parameters - a markov decision process and a value for the convergence threshold. We start by initialising the values of all states to be zero, and all other variables in its respective data structure (e.g. number of iterations ’num_iterations’ initialised to 0, and a Boolean variable ’converged’ that indicates convergence initialised to False) to facilitate the implementation of the algorithm. At each iteration, a variable is initialised to hold the maximum change in utility from the previous iteration. The Bellman Update iteration is then conducted for each state to calculate utility for the next state and set action that maximises expected utility. The maximum change in utility among all states is calculated and compared against the convergence threshold. When the maximum change in utility is less than the convergence threshold, the algorithm is deemed to have converged and the iteration stops.

The following presents the python implementation for the functions to get expected utility and perform bellman update iteration.

Listing 3: Python implementation of function to get expected utility

```python
def get_expected_utility(mdp, state, action, U_current):
    sum_expected_utility = 0
    next_states = mdp.get_next_states(state, action)

    for next_state in next_states:
        probability = mdp.transition_model(state, action, next_state)
        actual_next_state = next_states[next_state]["actual"]
        sum_expected_utility += probability * U_current[actual_next_state]

    return sum_expected_utility
```

Listing 4: Python implementation of function to perform bellman update iteration

```python
def bellman_update(mdp, state, U_current):
    max_expected_utility = float('-inf')
    best_action = None

    for action in mdp.actions:
        expected_utility = get_expected_utility(mdp, state, action, U_current)
        if expected_utility > max_expected_utility:
            max_expected_utility = expected_utility
            best_action = action

    utility = mdp.reward_function(state) + mdp.discount * max_expected_utility
    return (utility, best_action)
```

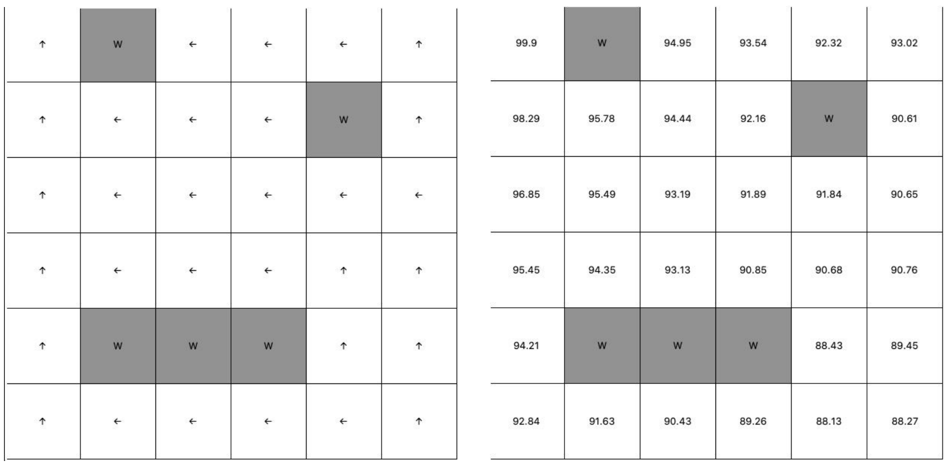
The following references code snippet from constants.py, the file responsible for the storage of all constant variables used in the program such as Discount Factor, Epsilon value, Reward Values, etc. This allows the configuration of environment settings of the experiment to be done easily by adjusting the values defined here.

Listing 5: Initialisation of Constants used in Value Iteration Algorithm

```python
## VALUE ITERATION ALGORITHM
C = 1
R_MAX = 1
EPSILON = C * R_MAX
CONVERGENCE_THRESHOLD = 0.07368
#CONVERGENCE_THRESHOLD = (EPSILON * (1 - DISCOUNT))/ DISCOUNT
```

3.3.4. Plot of Optimal Policy - Value Iteration (10 marks)

Implementation of the visualisation of the optimal policy was done through the Python Graphic User Interface (GUI) library Tkinter. This allows us to visualise the optimal policy and final utility values of all states in our grid environment. Full implementation can be found in the python file grid_visiualisation.py under the algorithm folder.



↑	W	←	←	←	↑
↑	←	←	←	W	↑
↑	←	←	←	←	←
↑	←	←	←	↑	↑
↑	W	W	W	↑	↑
↑	←	←	←	←	↑

(a) Plot of Optimal Policy

99.9	W	94.95	93.54	92.32	93.02
98.29	95.78	94.44	92.16	W	90.61
96.85	95.49	93.19	91.89	91.84	90.65
95.45	94.35	93.13	90.85	90.68	90.76
94.21	W	W	W	88.43	89.45
92.84	91.63	90.43	89.26	88.13	88.27

(b) Utilities of all states

Figure 5: Plot of Optimal Policy when C=0.10

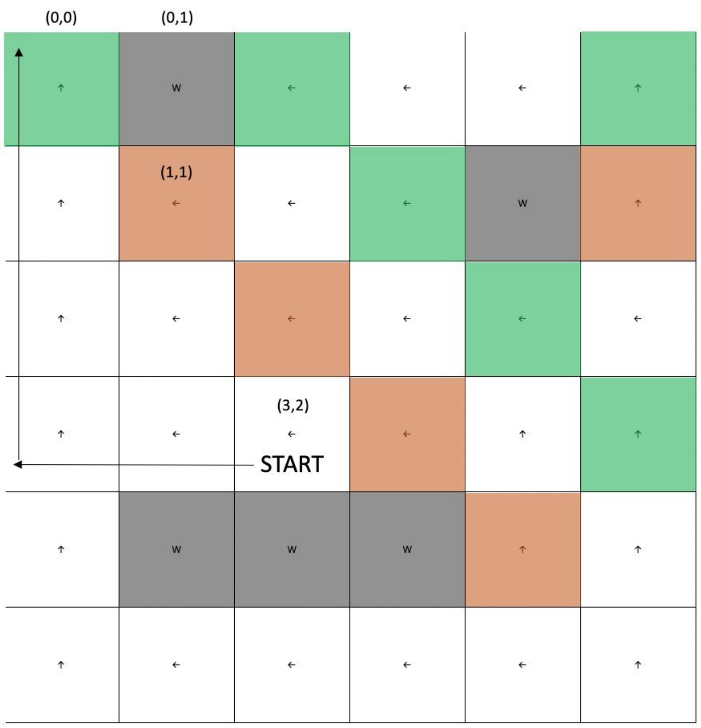


Figure 6: Optimal Policy - Value Iteration

From the plot of the optimal policy, the policy tends to recommend taking lefts and ups when $R(s)$=-0.04. As we track the agent from the starting point (3,2), the optimal policy recommends taking 2 steps left to (3,0), then 3 steps up to (0,0) to receive the +1 reward in order to maximise the utility of the subsequent states. As $R(s)$=-0.04 is a moderate cost (4% of 1) in relative to the +1 reward, the optimal policy reflects a more direct path to the reward by choosing to go towards (0,0). If the cost of one step were lower, the optimal policy would instead choose to be more conservative. For instance, the optimal policy could recommend taking a left at (1,0) to remove any possibility of falling into (1,1) which would incur a reward of -1. As the agent's state sequence is infinite, the discount factor γ=0.99 is introduced to help the utility stay bounded and make sure the algorithm converges. This results in the optimal policy favouring sooner rewards than later rewards. This is reflected in the plot as the policy recommends taking 2 lefts, then 3 ups to (0,0), instead of 2 lefts, 2 downs, 5 rights, then 2 up to reach (3,5), despite the latter route being less risky.

3.3.5. Utilities of all states (10 marks)

Listing 6: Utilities of all States - Value Iteration

```
**********************************
Value Iteration Algorithm
Values of Parameters
**********************************
Discount Factor         : 0.99
Utility Upper Bound     : 100
Max Reward(Rmax)        : 1
Constant 'c'            : 0.1
Epilson Value           : 0.1
Convergence Threshold   : 0.00101
Number of Iterations    : 688
**********************************
Utility Values of all States
**********************************
 State : Value
(0, 0) : 99.89968204081664
(0, 2) : 94.94513927496577
(0, 3) : 93.5394285674199
(0, 4) : 92.3219753688338
(0, 5) : 93.01717781580467
(1, 0) : 98.29304355148425
(1, 1) : 95.78269942578147
(1, 2) : 94.4446804097283
(1, 3) : 92.15634356630599
(1, 5) : 90.60922899068014
(2, 0) : 96.8481822226738
(2, 1) : 95.48610979237777
(2, 2) : 93.19410965534965
(2, 3) : 91.88642861836806
(2, 4) : 91.84254837288219
(2, 5) : 90.65374925556397
(3, 0) : 95.45352114231456
(3, 1) : 94.35217584302627
(3, 2) : 93.13222746277978
(3, 3) : 90.85069634200686
(3, 4) : 90.67765452334531
(3, 5) : 90.75899448238185
(4, 0) : 94.21220145276966
(4, 4) : 88.42533280100476
(4, 5) : 89.44975404856164
(5, 0) : 92.83715635781114
(5, 1) : 91.62845967056681
(5, 2) : 90.43483401433552
(5, 3) : 89.2560914710248
(5, 4) : 88.12866727276278
(5, 5) : 88.26630420296615
**********************************
```

3.3.6. Plot of utility estimates as a function of number of iterations (10 marks)

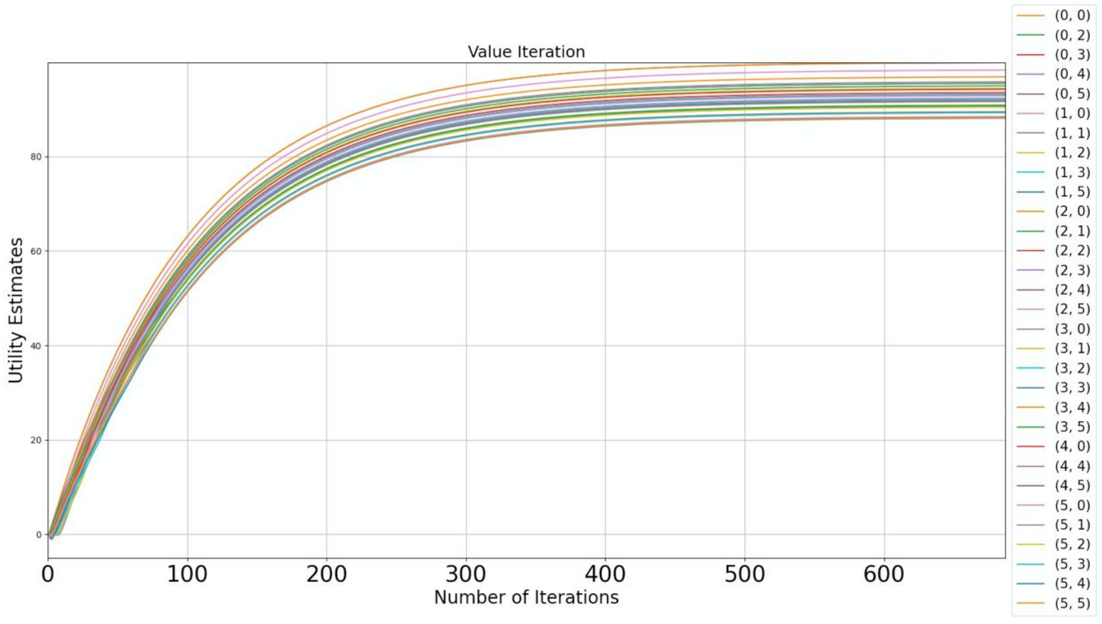


Figure 7: Plot of Utility Estimates as a function of the number of iterations (Value Iteration)

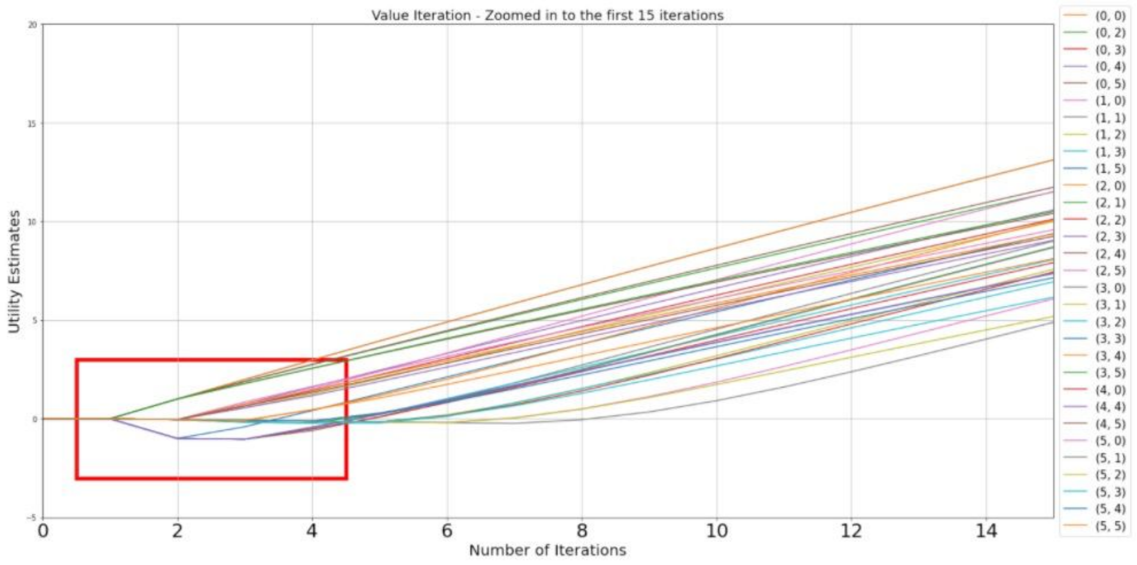


Figure 8: Plot of Utility Estimates as a function of the number of iterations (Value Iteration) - Zoomed in to the first 15 iterations

With reference to the graph above, we can see that at the start of the algorithm running, we observe that some of the utilities decrease and become negative, such as (4,4), before increasing once a path is found.

3.4. Policy Iteration

3.4.1. Theory of Policy Iteration Algorithm

The policy iteration algorithm is used to identify optimal policies without the need to solve the standard Bellman equations which the value iteration algorithm does. The policy iteration algorithm does this by alternating between 2 steps: Policy evaluation and Policy improvement iteratively and terminates when the policy improvement steps yields no change in the utilities - indicating convergence of a policy. This convergence is guaranteed to converge to the optimal policy which serves as the goal of the algorithm.

3.4.2. Implementation of Policy Iteration

Listing 7: Python implementation of Policy Iteration Algorithm

```python
def policy_iteration(mdp: MDP, k: int):
    utilities, policies = {}, {}

    # Supplementary dictionaries for visualisation
    U_iterations= {}

    for state in mdp.states:
        utilities[state] = 0
        policies[state] = mdp.actions[random.randint(0,3)] #the number will
            correspond to an action
        U_iterations[state] = [0]

    unchanged = False
    num_iterations = 0

    while not unchanged:
        utilities, new_iteration_utilities = policy_evaluation(policies,
            utilities, mdp, k)
        policies, unchanged = policy_improvement(mdp, utilities, policies,
            unchanged)
        num_iterations += int(k)
        print("Iteration: " + str(num_iterations))

        for state in mdp.states:
            U_iterations[state].extend(new_iteration_utilities[state])
```

The above code snippet describes the python implementation for the policy iteration algorithm. The function takes in 2 parameters - a markov decision process and a value k, which represents the number of times the process of policy evaluation is repeated to obtain an improved policy. In policy iteration, we start from an initial policy and alternate between the process of policy evaluation and policy improvement, until the process of process improvement yields no change in utilities.

We start by initialising the values of all states to be zero, and all other variables in its respective data structure (e.g. utility values of each states initialised to zero, a Boolean variable "unchanged" that indicates changes in policy initialised to False and an initial policy selected randomly) to facilitate the implementation of the algorithm. At each iteration, policy evaluation calculates utility for each state using the current policy. These values are passed onto policy improvement, which will return a new policy if there is an improved policy among all possible actions. When there is no change from policy improvement, the algorithm is deemed to have converged and the iteration stops.

The following presents the python implementation for the functions to perform policy evaluation and policy improvement.

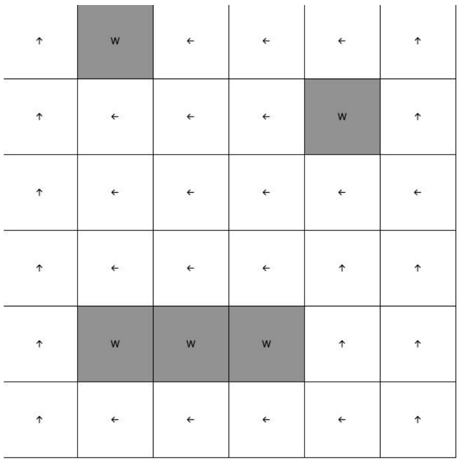
Listing 8: Python implementation of Policy Evaluation

```python
def policy_evaluation(policies, utilities, mdp, k):
    U_current, U_updated = {}, {}
    new_iteration_utilities = {}

    for state in mdp.states:
        U_current[state] = utilities[state]
        new_iteration_utilities[state] = []

    for i in range(int(k)):
        for state in mdp.states:
            # We use get_expected_utility to calculate  sP (  s  |s, a)U[ s ]
                where we replace a with the action under
            expected_utility = get_expected_utility(mdp, state,
                policies[state], U_current)

            # U_i+1(s)    R(s) +      sP    (s'|s,  _i (s)) U_i(s')
            U_updated[state] = mdp.reward_function(state) + mdp.discount *
                expected_utility

        for state in mdp.states:
            U_current[state] = U_updated[state]
            new_iteration_utilities[state].append(U_updated[state])

    return (U_current, new_iteration_utilities)
```

Listing 9: Python implementation of Policy Improvement

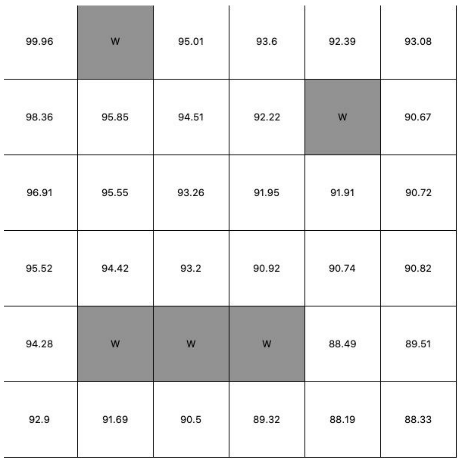
```python
def policy_improvement(mdp, utilities, policies, unchanged):
    updated_policies = policies

    unchanged = True

    for state in mdp.states:
        # 1. Find max     aA  (s) P( s   |s,a) U[ s ]
        max_expected_utility = float('-inf')
        best_action = None

        for action in mdp.actions:
            expected_utility = get_expected_utility(mdp, state, action,
                utilities)
            if expected_utility > max_expected_utility:
                max_expected_utility = expected_utility
                best_action = action

        # 2. Find    s' P( s   |s,  [s]) U[ s ]
        policy_utility = get_expected_utility(mdp, state, policies[state],
            utilities)

        # 3. Compare
        if max_expected_utility > policy_utility:
            updated_policies[state] = best_action
            unchanged = False

    return (updated_policies, unchanged)
```

3.4.3. Plot of Optimal Policy - Policy Iteration (10 marks)



↑	W	←	←	←	↑
↑	←	←	←	W	↑
↑	←	←	←	←	←
↑	←	←	←	↑	↑
↑	W	W	W	↑	↑
↑	←	←	←	←	↑

(a) Plot of Optimal Policy



99.96	W	95.01	93.6	92.39	93.08
98.36	95.85	94.51	92.22	W	90.67
96.91	95.55	93.26	91.95	91.91	90.72
95.52	94.42	93.2	90.92	90.74	90.82
94.28	W	W	W	88.49	89.51
92.9	91.69	90.5	89.32	88.19	88.33

(b) Utilities of all states

Figure 9: Plot of Optimal Policy when K=100

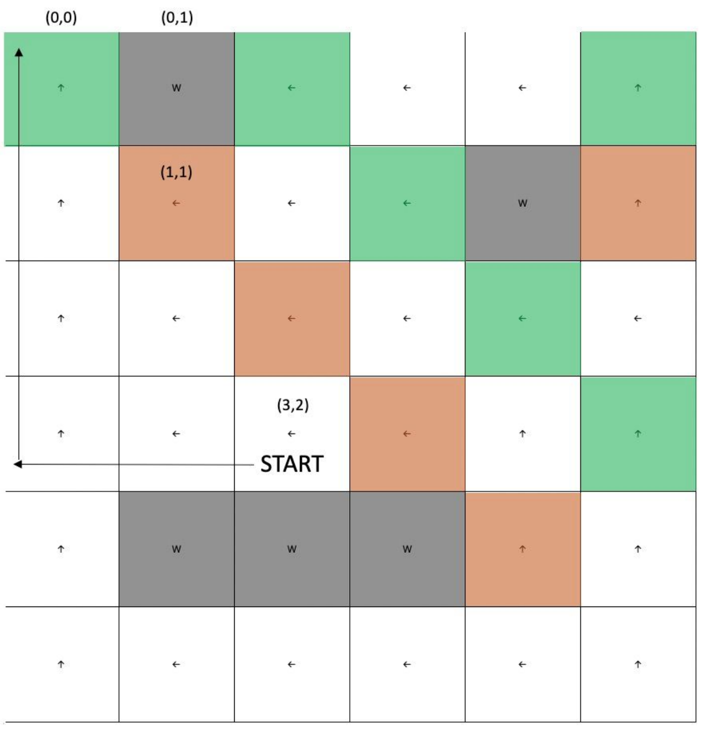


Figure 10: Optimal Policy - Policy Iteration

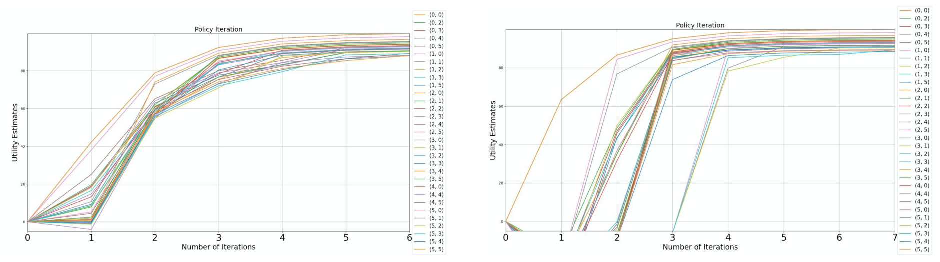
As expected, the resultant plot of the optimal policy is the same as the plot derived from the value iteration algorithm. However, this plot was attained with a much smaller number of iterations, at 7, as compared to under the value iteration algorithm, at 688 number of iterations.

3.4.4. Utilities of all states (10 marks)

Listing 10: Utilities of all States - Policy Iteration

```
***********************************
Policy Iteration Algorithm
Values of Parameters
***********************************
Discount Factor        : 0.99
K                      : 100
Number of Iterations   : 7
***********************************
Utility Values of all States
***********************************
 State : Value
(0, 0) : 99.83747800890235
(0, 2) : 94.88293524305142
(0, 3) : 93.47722453550556
(0, 4) : 92.25977133691948
(0, 5) : 92.95497378387992
(1, 0) : 98.23083951956995
(1, 1) : 95.72049539386717
(1, 2) : 94.382476377814
(1, 3) : 92.0941395343917
(1, 5) : 90.5470249587539
(2, 0) : 96.7859781907595
(2, 1) : 95.42390576046348
(2, 2) : 93.13190562343537
(2, 3) : 91.82422458645375
(2, 4) : 91.7803443409679
(2, 5) : 90.59154522364818
(3, 0) : 95.39131711040025
(3, 1) : 94.28997181111197
(3, 2) : 93.07002343086548
(3, 3) : 90.78849231009261
(3, 4) : 90.61545049143085
(3, 5) : 90.696790450466
(4, 0) : 94.14999742085539
(4, 4) : 88.36312876909005
(4, 5) : 89.38755001664579
(5, 0) : 92.77495232589686
(5, 1) : 91.56625563865256
(5, 2) : 90.37262998242126
(5, 3) : 89.19388743911053
(5, 4) : 88.06646324084848
(5, 5) : 88.20410017105024
***********************************
```

3.4.5. Plot of utility estimates as a function of number of iterations (10 marks)



(a) Run 1 (b) Run 2

Figure 11: Plot of utility estimates as a function of number of iterations - 2 runs

From the graphs of utility estimates against number iterations using the policy iteration algorithm, we observe that there is a similar general trend of increasing steadily then converging to a value. However, the number of iterations needed to reach convergence are 100x lower than value iteration. Another interesting observation is that although the two graphs above were run consecutively with the same parameters, the graphs generated share rather distinct differences and converge at different number of iterations - 6 and 7 respectively. This is in line with the theory of the policy iteration algorithm, where the initial values for utilities begin at random value - causing the number of iterations needed for convergence to vary even with the same parameters.

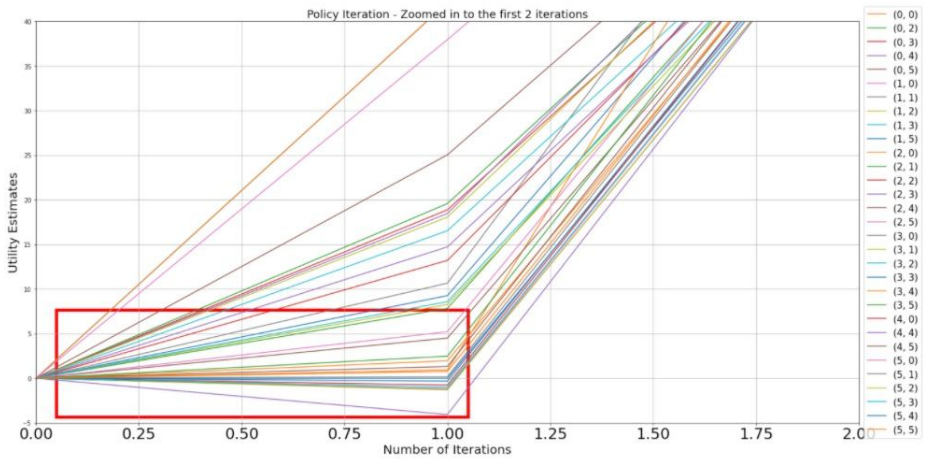


Figure 12: Plot of Utility Estimates as a function of the number of iterations (Policy Iteration) - Zoomed in to the first 15 iterations

Similar to the value iteration algorithm, as we zoom in to the first few iterations of the policy iteration algorithm, we observe that some of the utilities decrease and become negative at the start of the algorithm - such as (2,3), before increasing once a path is found.

4. Bonus Question

4.1. Design of a maze environment with increased total number of states and complexity

A maze environment with increased total number of states can be implemented by increasing the number of rows and columns during maze creation. On top of that, a maze environment with varying complexity can be implemented by adjusting the probability of the types of squares assigned to individual states during maze creation. For example, by increasing the probability of a state to be a "Green" square, we introduce more squares with +1 reward into the agent environment - thereby increasing the maze complexity.

Listing 11: Python implementation to generate more complex maze

```python
def generate_maze(grid_length):
    random.seed()
    maze = []

    for row in range(grid_length):
        maze.append([])

        for _ in range(grid_length):
            random_color = random.random()
            if random_color < WHITE:
                maze[row].append(' ')
            elif random_color < WHITE + GREEN:
                maze[row].append('G')
            elif random_color < WHITE + GREEN + BROWN:
                maze[row].append('B')
            else:
                maze[row].append('W')
    return maze
```

With reference to the code snippet from bonus_functions.py above, a function generate_maze(grid_length,p_white,p_green,p_brown,p_wall) is used to create a maze environment. It takes in an integer grid length as its parameter, which is used to configure the size of the maze to be created. In order to determine which squares are Green (+1 reward), Brown (-1 Reward), White (Blank, -0.04 Reward) and Gray (Wall), individual probabilities for the different states are defined globally which are used for assignment to individual squares during maze creation. This implementation allows us to vary both the size of the maze, as well as the complexity of the maze independently.

4.2. How does the size of the environment (total number of states) affect convergence

In order to understand the relationship between the number of states in our environment and convergence of optimal policy algorithms, the following experiments were set up to vary the size of the maze generated, while keeping the complexity of the maze constant.

4.2.1. Number of environment states affecting time to convergence

To investigate the relationship between the number of states in our environment and the time taken for the algorithms to converge, each maze was put through the same optimal policy algorithm (value and policy iteration) - recording the time taken for each algorithm to run, which serves as proxy for time to convergence.

Listing 12: Python implementation to extract time and number of iterations taken for convergence with increasing maze size

```python
def vary_mazesize(algorithm):
    size_of_maze=[]
    run_times=[]
    iterations_to_convergence=[]

    for i in range(6,30):
        bigger_grid = generate_maze(i)
        maze3a = Maze(bigger_grid, REWARD_MAP, DISCOUNT)
        start = time.time()

        if (algorithm=="valueiteration"):
            #  Run Value Iteration Algorithm
            value_iteration_res = value_iteration(maze3a)
            num_iterations = value_iteration_res['num_iterations']
        elif (algorithm=="policyiteration"):
            # Run Policy Iteration Algorithm
            policy_iteration_res = policy_iteration(maze3a, K)
            num_iterations = policy_iteration_res['num_iterations']

        end = time.time()
        size_of_maze.append(i)
        run_times.append(end-start)
        iterations_to_convergence.append(num_iterations)
```

With reference to the implementation above, the function vary_mazesize(algorithm) in the bonus_functions.py file creates a maze with increasing size and puts them through an optimal policy algorithm. The time taken to convergence for each maze is then recorded for visualisation below. Also, due to the compute limitation of our local machines used to run the programs, maze sizes from 6 to 29 has been used to investigate this relationship. To account for the variability of individual trials, the above experiment was ran a total of 100 times and an average was taken to obtain the average time taken for each algorithm to run with each size of the maze. With the results obtained, the following plots illustrate the trend of the time to convergence with respect to maze size for both our value iteration and policy iteration algorithms.

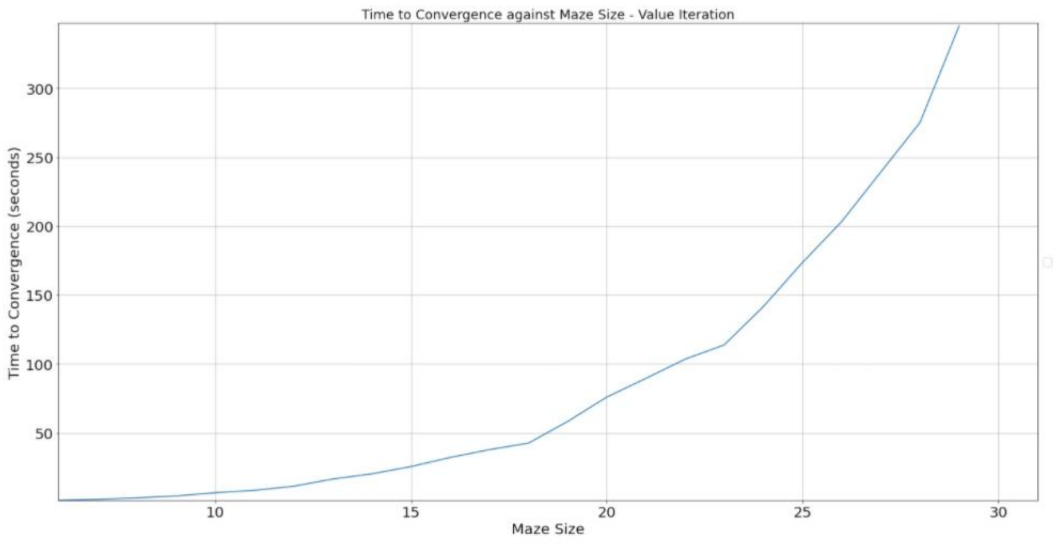


Figure 13: Plot of Time to Convergence against Maze Size (Value Iteration)

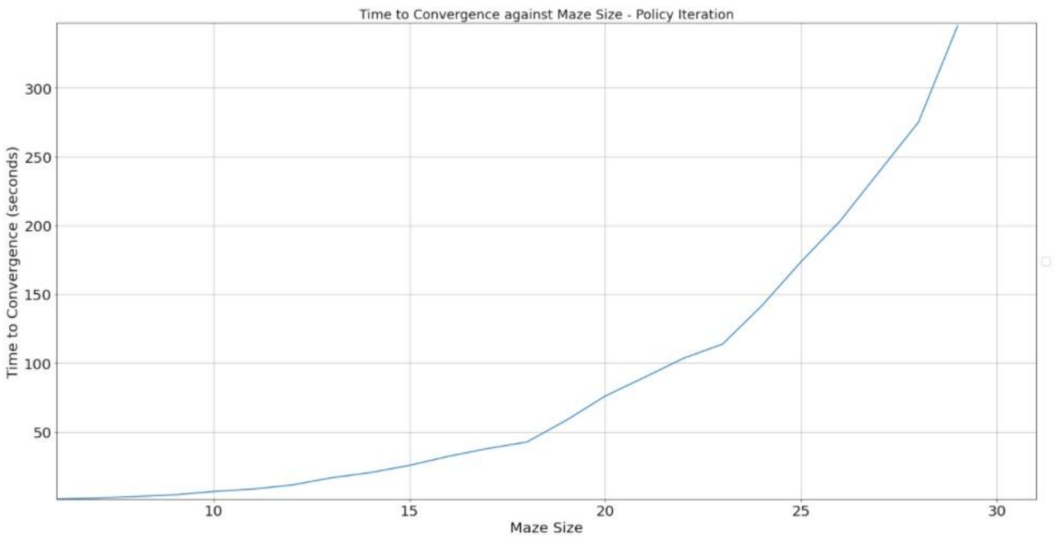


Figure 14: Plot of Time to Convergence against Maze Size (Policy Iteration)

With reference to the figures above, we can see that the graphs for value iteration and policy iteration algorithms are almost identical. As the maze size increases from 6 to 29, the time taken for convergence can be observed to increase non-linearly.

4.2.2. Number of environment states affecting number of iterations to convergence

To investigate the relationship between the number of states in our environment and the number of iterations needed for each algorithm to converge, each maze was put through the same optimal policy algorithm (value and policy iteration) - this time, the number of iterations needed for each algorithm to complete is recorded.

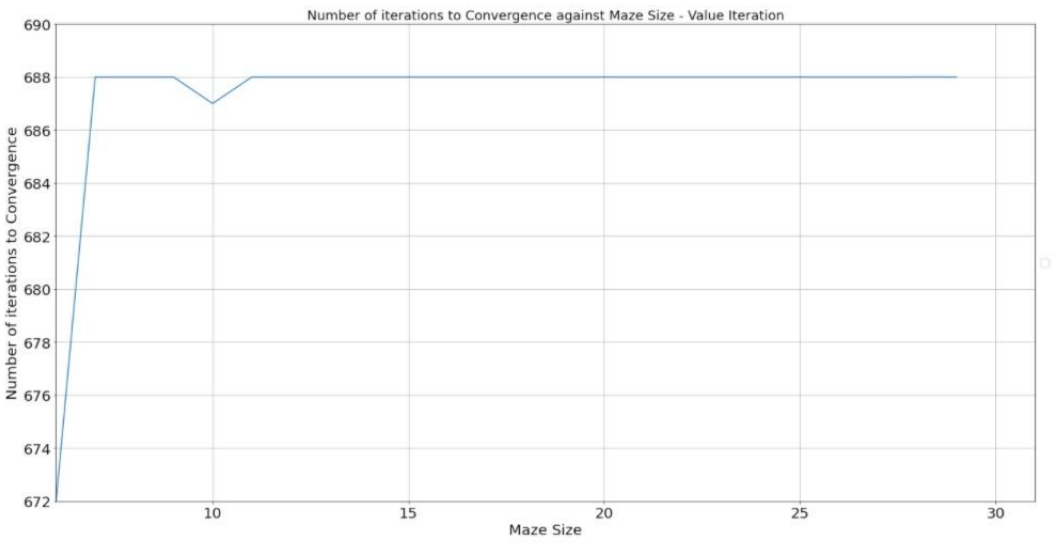


Figure 15: Plot of Iterations to Convergence against Maze Size (Value Iteration)

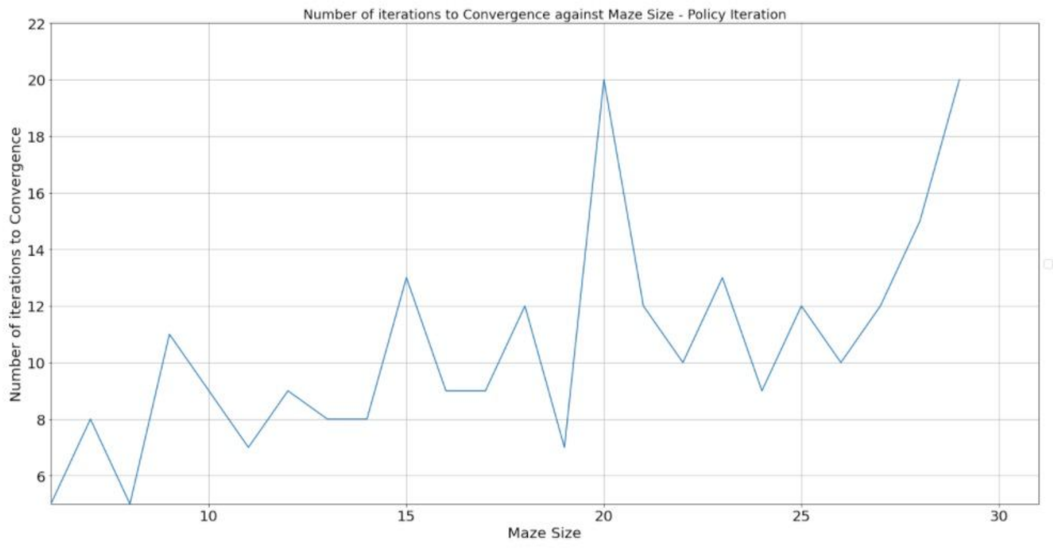


Figure 16: Plot of Iterations to Convergence against Maze Size (Policy Iteration)

Unlike the graphs we have obtained for time taken to convergence, the relationships between the number of iterations to convergence against maze size is drastically different for the value iteration and policy iteration algorithm. For the value iteration algorithm, we can see that the number of iterations needed to converge stays at 688 despite an increasing maze size from 6 to 29. Although there is a noticeable dip of number of iterations to 687 when the maze size is 10, that could be due attributed to a locality variance, since the rest of the graph falls consistently at 688. This tells us that the number of iterations for the value iteration algorithm to converge is independent of the total number of states in the environment.
On the other hand, we can see that the relationship between the number of iterations to convergence against maze size for the policy iteration algorithm is rather sporadic.

4.3. How does the complexity of the environment affect convergence

In order to understand the relationship between the complexity of agent environment and convergence of optimal policy algorithms, the following experiments were set up to vary the probabilities of the types of squares assigned to individual states during maze creation, while keeping the size of the maze constant at 10x10 for computational efficiency.

Unlike maze size where there is a clear definition to quantify the particular property (E.g. number of rows, number of columns and number of total states), there is a lack of a clear definition we can use to define complexity of the maze. Hence, there is a need for us to outline a heuristic to indicate complexity as we investigate this relationship. For this experiment, we define the complexity of the maze as the probability of a square being a green square. In order to keep the total number of states constant, the probability of a square being a wall has been set as a constant at 0.15. The remaining number of squares will be split among an brown and a white (blank) square with a fixed proportion of 0.3:0.7.

Listing 13: Python implementation to generate maze with varying complexity

```python
def generate_maze(grid_length, p_green):
    GREEN = p_green
    WALL = 0.15
    BROWN = 0.3*(1-GREEN-0.15)
    WHITE = 0.7*(1-GREEN-0.15)
...
...
```

With reference to the above code snippet, the function generate_maze(grid_length,p_green) has been modified slightly to accommodate the need to adjust the probabilities of the different types of squares based on the probability of the green square. This time, it takes in an additional parameter p_green, which is used to allocate the probabilities of the other types of squares during maze creation.

Listing 14: Python implementation to extract time and number of iterations taken for convergence with varying maze complexity

```python
def vary_complexity(algorithm):
    p_green_list=[]
    run_times=[]
    num_iterations_list=[]
    for p_green in numpy.arange(0,1.1,0.1):
        bigger_grid = generate_maze(10,p_green)
        maze3a = Maze(bigger_grid, REWARD_MAP, DISCOUNT)
        start = time.time()

        if (algorithm=="valueiteration"):
            #  Run Value Iteration Algorithm
            value_iteration_res = value_iteration(maze3a)
            num_iterations = value_iteration_res['num_iterations']
        elif (algorithm=="policyiteration"):
            # Run Policy Iteration Algorithm
            policy_iteration_res = policy_iteration(maze3a, K)
            num_iterations = policy_iteration_res['num_iterations']

        end = time.time()
        p_green_list.append(p_green)
        run_times.append(end-start)
        num_iterations_list.append(num_iterations)
```

With the implementation above, we increase the complexit of the maze by varyning the probability p_green from 0 to 1, with each step of 0.01. With each value of p_green, we generate a maze and have it put through the same optimal policy algorithms (value and policy algorithm) - recording both the time taken and number of iterations taken for the algorithm to converge.

4.3.1. Complexity of environment affecting time to convergence

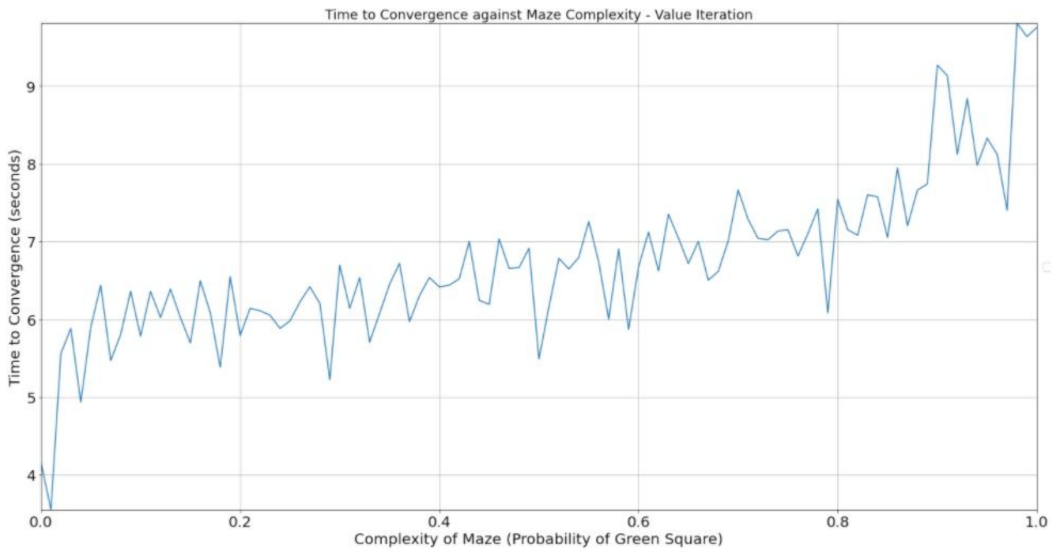


Figure 17: Plot of Time taken to Convergence against Maze Complexity (Value Iteration)

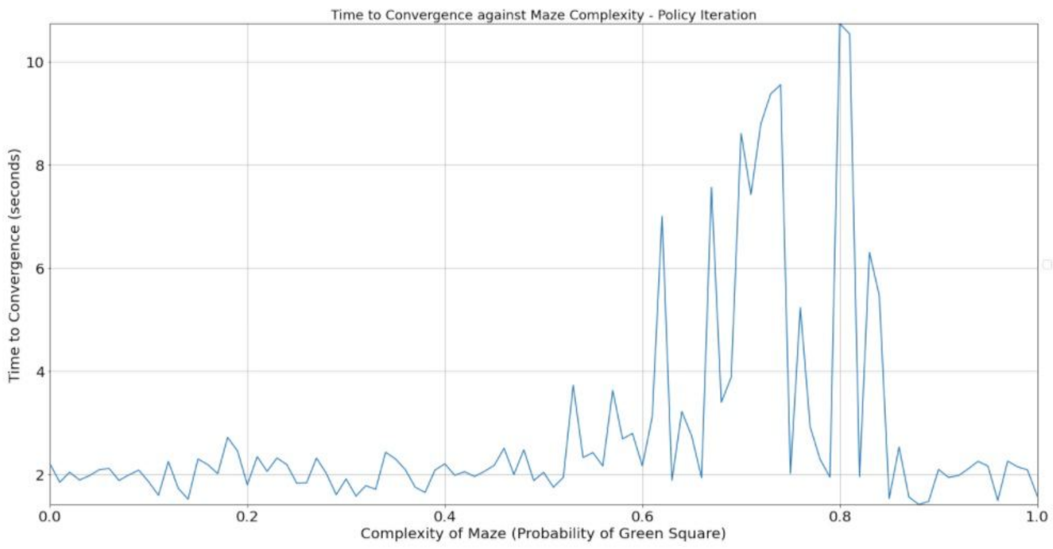


Figure 18: Plot of Time taken to Convergence against Maze Complexity (Policy Iteration)

With reference to the graphs above, we observe that as the maze gets increasingly more complex, the time taken for the value iteration algorithm to converge increases from about 5 seconds at p_green=0.10 to about 10 seconds when p_green=1.0. We should also note that with p_green at the range of 0.2-0.8, the time taken for the algorithm to converge increases only so slightly from around 6 seconds to 7 seconds.

On the other hand, we can see that the policy iteration algorithm behaves differently with increased maze complexity. As the complexity of the maze is increased with p_green at 0 to p_green at 0.6, the time taken for the algorithm to converge is rather stable at 2 to 3 seconds. However, at the range of p_green between 0.6 and 0.8, we can observe a spike in time taken for convergence to about 10-12 seconds. When p_green is larger than 0.8, the time taken for the algorithm to converge falls back to the range of 2 to 3 seconds.

4.3.2. Complexity of environment affecting number of iterations to convergence

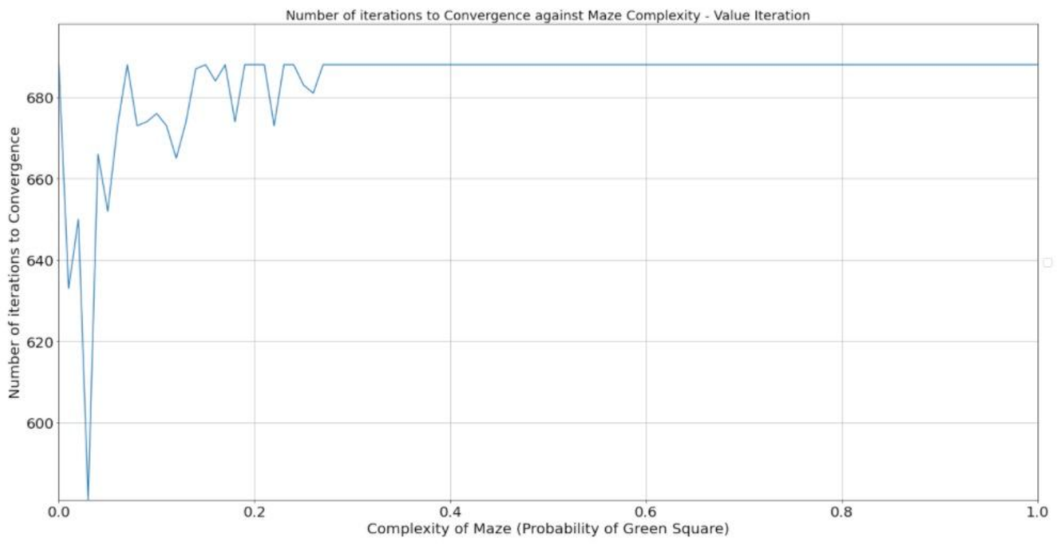


Figure 19: Plot of Iterations to Convergence against Maze Complexity (Value Iteration)

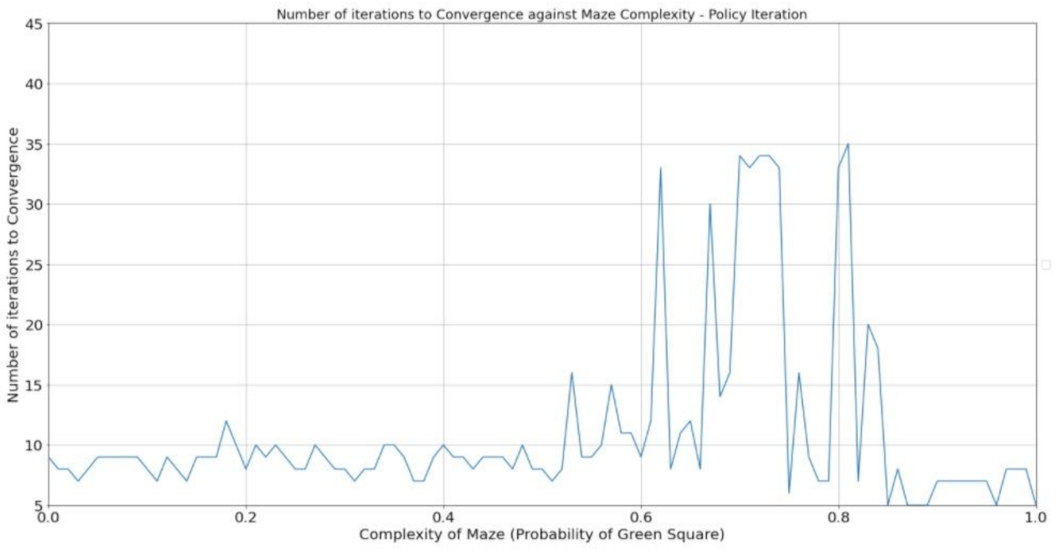


Figure 20: Plot of Iterations to Convergence against Maze Complexity (Policy Iteration)

With reference to the graphs above, we observe that as the maze gets increasingly more complex with p_green at 0 to 0.2, the number of iterations it takes for the value iteration algorithm to converge increases rapidly from about 550 iterations to about 680 iterations. When p_green is increased beyond 0.25, we observe that the number of iterations for the algorithm to converge stabilises at 688.

On the other hand, we can see that the policy iteration algorithm behaves differently with increased maze complexity. Similar to time taken for convergence with increased maze complexity, the number of iterations taken for the algorithm to converge is rather stable at about 10 iterations when p_green is increased from 0 to 0.6. At p_green between 0.6 to 0.8, we can similarly observe a spike in the number of iterations taken for convergence to 30-35 iterations. When p_green is larger than 0.8, the number of iterations for the algorithm to converge falls back to the range of about 10 iterations.

5. Closing

In this paper, we implemented the Grid World problem in order to investigate the behaviour and performance of 2 optimal policy algorithms - value iteration and policy iteration, in a given Markov Decision Process environment. By varying the agent environment both in terms of size and complexity (outlined in Bonus question), we observe some intuitive results with reference to theoretical mechanisms of the algorithm. For example, we observed that for value iteration algorithm, the number of iterations to convergence is a function of the convergence threshold value, and is not affected by the size of the maze. Additionally, the entire experiment demonstrated that the number of iterations taken for the policy iteration algorithm to converge is substantially (100x) lower than the number of iterations taken for the value iteration algorithm. However, based on research beyond the scope of the paper, each iteration of the policy iteration algorithm is found to be more computationally expensive than an iteration of the value iteration algorithm - which presents an important point of consideration moving forward.