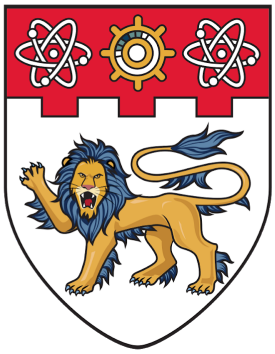


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

Student Details

Name	Acharya Atul
Matriculation Number	U1923502C
Email	ATUL001@e.ntu.edu.sg
Course Code	CZ4046
Course Name	Intelligent Agents

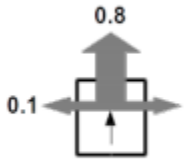
1. Problem Overview

Given a maze environment, convert it to an MDP and solve using both value iteration and policy iteration.

1.1. Problem Description

Maze Environment

The maze environment is a 6x6 grid world which contains walls, rewards and penalties. All green squares have a reward of +1. All orange/red squares have a penalty of -1. All white squares have a reward of -0.04. All grey squares are walls and these are unreachable states. Hence the reward for walls need not be defined and has been set to 0 for this experiment.



Transition Model

The transition model is as follows: 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. The rewards for the white squares are -0.04, for the green squares are +1, and

for the brown squares are -1. Note that there are no terminal states; the agent's state sequence is infinite.

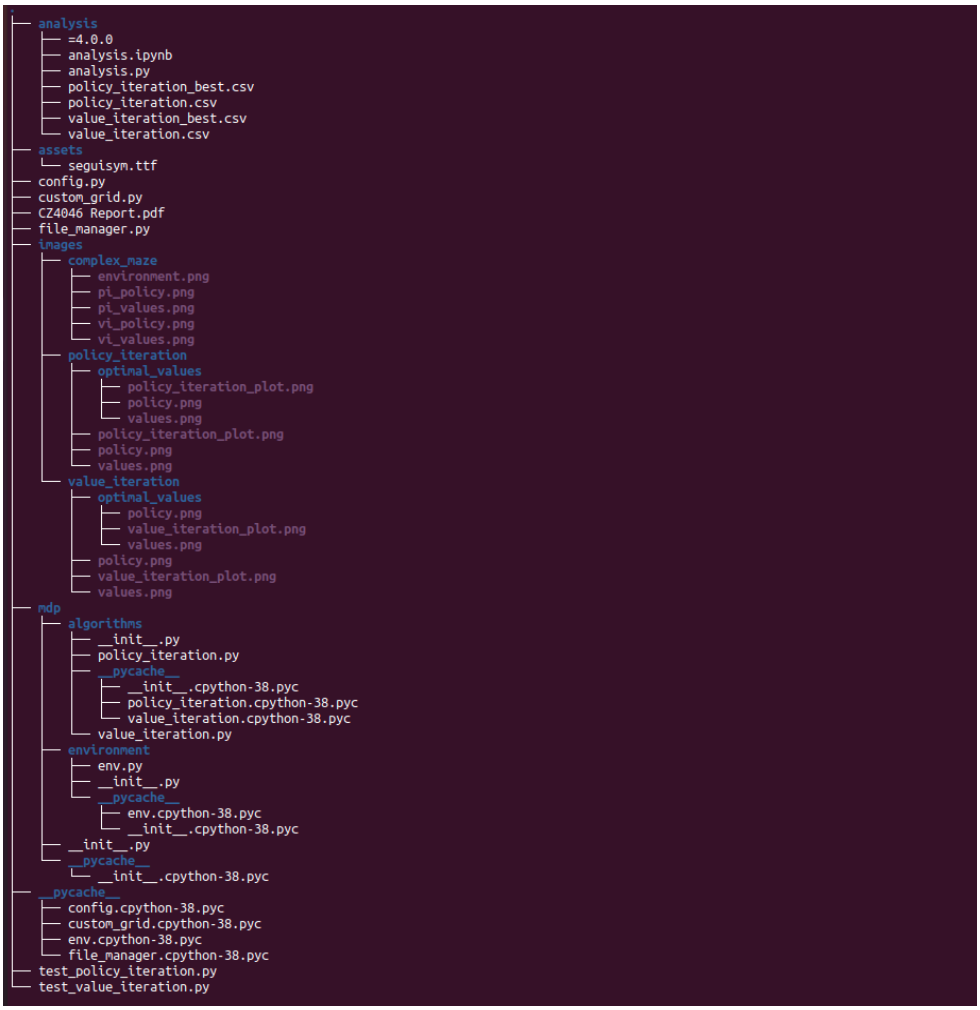
1.2. Expected Solution

From the grid world description, the agent will try to transition to state (0, 0). In this state the agent will choose the action NORTH (goes up). There are walls on all 3 possible directions that the agent can transition to and hence the agent will remain in the same state.The reward for this state is +1. Hence, the agent can accumulate infinite reward by staying in state (0, 0).

2. File Structure and Details

2.1. File Tree

The project folder contains 5 python files and 2 packages. The 5 python files contain code to test the algorithms, configure the maze environment and save data as a .csv file. The project folder consists of 2 other packages - mdp and analysis. The mdp package contains 2 packages - algorithms and environment. The algorithms package contains 2 python files which implement the value and policy iteration algorithms. The environment package contains 1 python file that contains the environment and stores it as an MDP. The analysis package contains a python notebook and 2 .csv files which renders the plot of estimated utilities over number of iterations.



Project File Tree

2.2. FIle Details

env.py: Contains the Environment class which stores information about the MDP.

policy_iteration.py: Contains the PolicyIteration class which implements the policy iteration algorithm to solve the MDP.

value_iteration.py: Contains the ValueIteration class which implements the value iteration algorithm to solve the MDP.

filemanager.py: Saves the utilities of every state calculated during the execution of the algorithms in each iteration as a .csv file.

config.py: Contains constants for initializing the MDP.

custom_grid.py: Contains variables for users to create a custom environment.

test_value_iteration.py: Driver program to test the value iteration algorithm.

test_policy_iteration.py: Driver program to test the policy iteration algorithm.

analysis.ipynb: Plots the utilities of every state vs iterations for both algorithms.

3. Implementation of MDP

3.1. Grid World

The grid world is a 6x6 array which contains walls, rewards and penalties. It is implemented as a 2-Dimensional list as shown below.

```
grid = [
    ['G','W','G','','', 'G'],
    ['','R','','G','W', 'R'],
    ['','','R','','G', ''],
    ['','','','R','', 'G'],
    ['','W','W','W','R', ''],
    ['','','','','','', ''],
]
```

Grid World representation in code

Cells marked as ‘G’ have a reward of +1. Cells marked as ‘R’ have a penalty of -1. Cells marked as ‘W’ are walls which are unreachable states. Cells that are marked with an empty string are normal cells which have a penalty of -0.04.

3.2. Actions

The possible actions that the agent can take are [UP, DOWN, LEFT RIGHT]. Each of these actions is represented by a tuple of 2 elements. The first element indicates the direction moved along the vertical direction and the second element indicates the direction moved along the horizontal direction.

The tuples of every state are given below

Action	**Tuple**
UP	(-1, 0)
DOWN	(+1, 0)
LEFT	(0, -1)
RIGHT	(0, +1)

To explain the reason for representing actions as tuples consider the Current State of the agent to be (1,1). Each state is represented in a (row, column) format.

If the agent moves UP, the next state of the agent should be (0, 1).
Current State + UP = (1, 1) + (-1, 0) = (0, 1)

If the agent moves DOWN, the next state of the agent should be (2, 1).
Current State + DOWN = (1, 1) + (+1, 0) = (2, 1)

If the agent moves LEFT, the next state of the agent should be (1, 0).
Current State + UP = (1, 1) + (0, -1) = (1, 0)

If the agent moves RIGHT, the next state of the agent should be (1, 2).
Current State + UP = (1, 1) + (0, +1) = (1, 2)

3.3. Rewards

In this problem, the reward only depends on the current state. Therefore, the rewards can be stored the same as the grid world.

Every green colored cell has a reward of +1. Every red colored cell has a reward of -1. Every white cell has a penalty of -0.04. The reward of the wall states does not matter since it is an unreachable state.

The reward is stored as a 2-Dimensional List as follows.

```
reward_map = {'G': +1, 'R': -1, 'W': 0, '': -0.04}

rewards = [[reward_map[cell] for cell in row] for row in grid]
```

Rewards Represented in Code

3.4. Transition Probabilities

The transition model is as follows: 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.

The transition probability is modeled as a function in the Environment Class. The function takes 2 arguments - current state and action and returns the transition probabilities.

```
def transition_model(self, state, action):
        model = {}
        possible_directions = [action, (action[1], action[0]), (-action[1], -action[0])]
        probability = [0.8, 0.1, 0.1]
        for idx, direction in enumerate(possible_directions):
            next_state = (state[0] + direction[0], state[1] + direction[1])
            if 0 <= next_state[0] < self.grid_width and 0 <= next_state[1] < self.grid_height:
                if self.grid_world[next_state[0]][next_state[1]] == 'W':
                    next_state = state
            else:
                next_state = state
            if next_state in model:
                model[next_state] += probability[idx]
            else:
                model[next_state] = probability[idx]
        return model
```

Transition Model Represented in Code

4. Value Iteration

The algorithm aims to calculate the utility of every state. Once the utilities converge, the algorithm computes the optimal policy by taking actions that maximize expected utility.

The utility of a state is the immediate reward for that state plus the expected discounted utility of the next state, assuming that the agent chooses the optimal action.

The starting utility of every state is initialized to 0 which are then updated using the Bellman equations. The Bellman equation relates the utility of a state to the utility of neighbouring states. The Bellman Update is performed until the utility of every state converges.

The Bellman update is given by the following formula

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

4.1. Pseudo-code for Value Iteration

The Pseudo-code from “Artificial Intelligence: A Modern Approach” by S. Russell and P. Norvig. Prentice-Hall, third edition, 2010 was used to implement the algorithm.

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

Pseudo-code followed

4.2. Implementation

The Value Iteration algorithm is implemented in value_iteration.py. When test_value_iteration.py is run, the following steps occur:

1. The environment (grid world) is created
2. Constants for Value Iteration algorithm are initialized
3. The MDP is solved using Value Iteration
4. A plot of the optimal policy and final utilities is displayed
5. The data required for analysis is saved as a .csv file

A set of constants are fixed in test_value_iteration.py

GAMMA (Discount Factor)	0.99
C	0.1
MAX_REWARD (Rmax)	1.0

The discount factor is set to 0.99. The maximum reward for any set is +1. The value of C is set to 0.1.

Epsilon is calculated using these values. Epsilon is the maximum error allowed in the utility of any state.
*epsilon = C * Rmax = 0.1 * 1 = 0.1*

The threshold is calculated using the formula:
*threshold = epsilon * (1 - gamma) / gamma = 0.1 * (1 - 0.99) / 0.99 = 0.00101*

A 2-Dimensional array is used to store the utility of every state. It is initialized to 0 and every cell is updated using the Bellman update

4.2.1. Bellman Update

The solve method of the Value Iteration Class implements the value iteration algorithm. It loops through all the states and updates the expected utility of every state using the bellman update. "*action_values*" is a list that stores the expected utility for every possible action. The *state_value* is updated using the statement:

*state_value = reward + gamma * max (action_values)*

Below is a code snippet that implements the Bellman update. The following Bellman update is performed for every state. The algorithm loops through all the states and the following code is a wrapper in this loop.

```
for action in actions:

    action_value = 0

    # Get the transition model
    transition_model = mdp.transition_model(cur_state, action)

    # Loop through all the next states in the transition model
    # This loop calculates expected utility for taking the action
    for next_state in transition_model:
        utility = utilities[next_state[0]][next_state[1]]
        probability = transition_model[next_state]
        expected_utility = probability * utility
        action_value += expected_utility

    action_values.append(action_value)

# Reward for the current state
reward = mdp.receive_reward(cur_state)

# Bellman Update
state_value = reward + self.gamma * max(action_values)
new_utilities[i][j] = state_value

# Update the value of delta
delta = max(delta, abs(new_utilities[i][j] - utilities[i][j]))
```

Bellman Update for each state

4.2.2. Convergence Condition

The algorithm runs until the utilities of every state converges. The delta value is originally set to 0 and is updated as:

delta = max (delta, new_utilities(state) - old_utilities(state))

The values converge once delta is less than the threshold (0.00101).

Below is a code snippet that shows the stopping condition of the algorithm.

```
# Check for convergence
if delta < threshold:
    break
```

Convergence Criteria

4.2.3. Calculation of Optimal Policy

Once the utilities of every state converge, the algorithm can then find the optimal policy. The optimal policy is calculated by selecting the action that maximizes the expected utility. The optimal policy is calculated in the greedify method of the Value Iteration Class. For every state the action value for every action is calculated and the action that has the action value is deemed to be the best action to take for that state.

The following code snippet calculates the optimal policy. The method loops through every state and selects the best action using the code snippet as shown below.

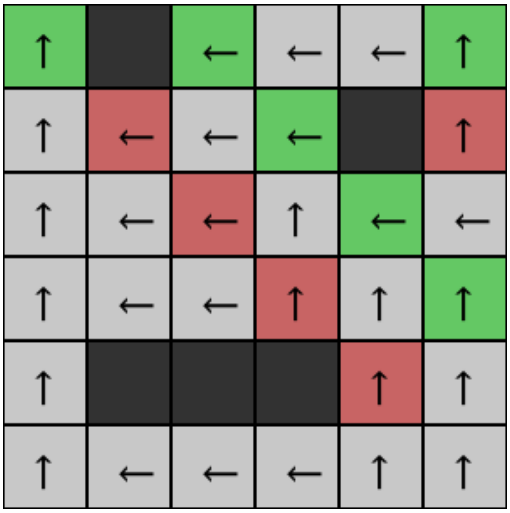
```
# Loop through all actions
for action in actions:
    action_value = 0
    # Get transition model
    transition_model = mdp.transition_model(cur_state, action)

    # Loop through all the next states in the transition model
    # Calculate the expected utility for taking the action
    for next_state in transition_model:
        utility = utilities[next_state[0]][next_state[1]]
        probability = transition_model[next_state]
        expected_utility = probability * utility
        action_value += expected_utility
    action_values[action] = action_value
best_action = None
best_action_value = -math.inf

# Choose best action
for action in action_values:
    if action_values[action] > best_action_value:
        best_action = action
        best_action_value = action_values[action]
policy[i][j] = best_action
```

Greedification to select optimal actions

4.3. Final Utilities and Optimal Policy

Plots of final utilities and optimal policy are shown below.

Plot of Final Utilities



Plot of Optimal Policy

The final utilities when C = 0.1 are shown. Every state is represented as (col, row).

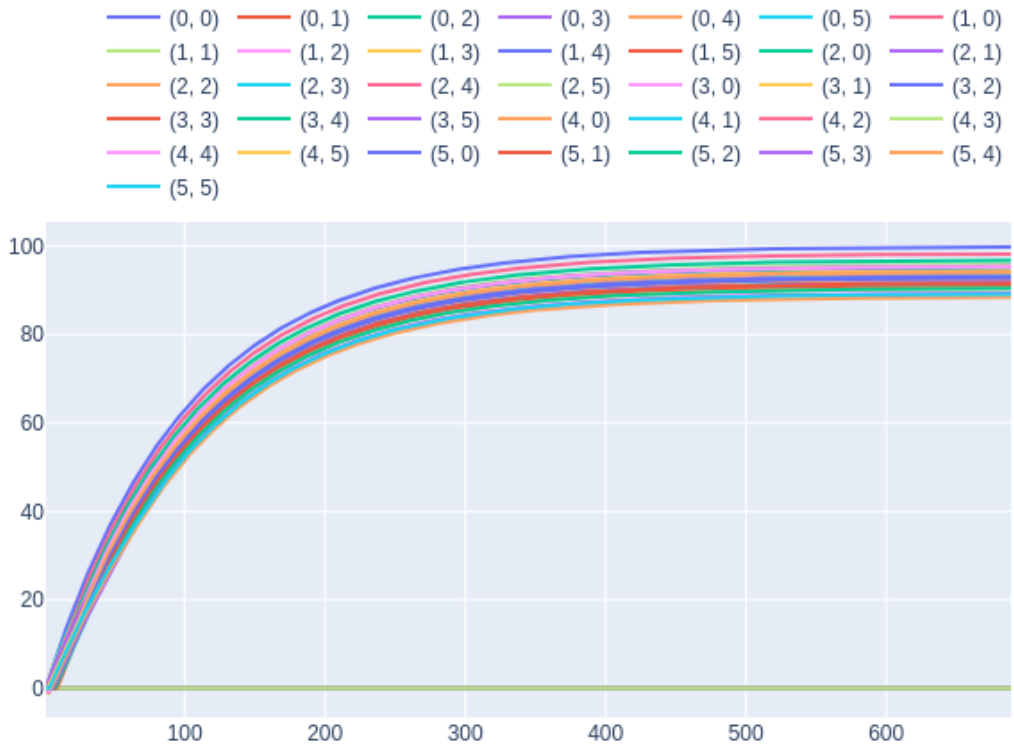
Number of iterations: 688

(0, 0): 99.9006852204078
(1, 0): 98.29404673107541
(2, 0): 96.84918540226495
(3, 0): 95.45452432190575
(4, 0): 94.21320463236087
(5, 0): 92.83815953740235
(0, 1): 0.0
(1, 1): 95.78370260537261
(2, 1): 95.48711297196897
(3, 1): 94.35317902261747
(4, 1): 0.0
(5, 1): 91.62946285015803
(0, 2): 94.94614245455695
(1, 2): 94.44568358931949
(2, 2): 93.19511283494086
(3, 2): 93.13323064237099
(4, 2): 0.0
(5, 2): 90.43583719392673
(0, 3): 93.77568621731811

(1, 3): 94.29840005646918
(2, 3): 93.07695824252093
(3, 3): 91.01594175092443
(4, 3): 0.0
(5, 3): 89.25709465061603
(0, 4): 92.55529967122197
(1, 4): 0.0
(2, 4): 93.00305429226383
(3, 4): 91.71509229342458
(4, 4): 89.44909832058741
(5, 4): 88.46978429758704
(0, 5): 93.22918825371615
(1, 5): 90.8186084153956
(2, 5): 91.69555629320722
(3, 5): 91.7887697818793
(4, 5): 90.46745089144353
(5, 5): 89.19837580875523

4.4. Plot of Estimated Utilities vs Iterations

A plot of Utility estimates as a function of iterations is shown below



Plot of utility estimates vs iterations

The above graph was plotted with C = 0.1. As the number of iterations increases, the utility estimates also increases. However, the utility estimates slowly converge and the algorithm takes 688 iterations to find the optimal policy.

From the above plots, it is clear that the algorithm has found the correct policy which is to reach the state (0, 0). The utility of state (0, 0) is also the highest.

4.5. Calculation of best value for C

In order to reduce the number of Bellman updates required to find the optimal policy, the value of C was varied until the algorithm could not find the optimal policy

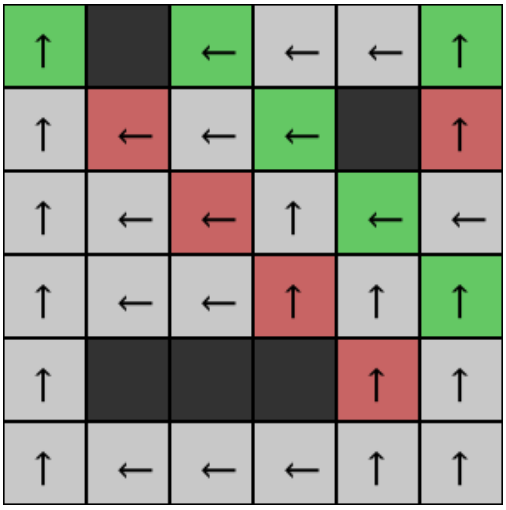
Results:

C	Iterations	Optimal Policy
0.1	688	Yes
1	459	Yes
10	230	Yes
25	138	Yes
40	92	Yes
45	80	Yes
50	69	No

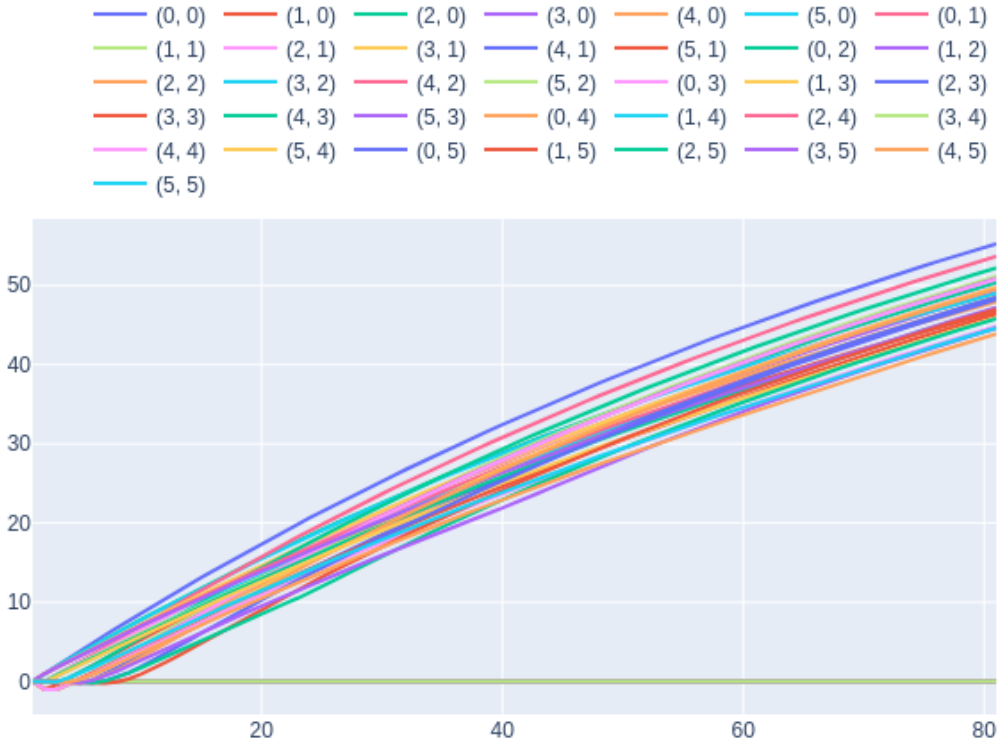
Plots:

Here are the corresponding plots of optimal policy and final utilities with C = 45

Plot of Final Utilities



Plot of Optimal Policy



Plot of utility estimates vs iterations

5. Policy Iteration

The algorithm aims to calculate the optimal policy. The algorithm starts with an initial policy and alternates between 2 steps: policy evaluation and policy improvement.

The policy improvement step calculates the utilities of every state using the current policy. Once the Bellman update is performed ‘k’ times, the algorithm moves on to the policy improvement step. The current policy is updated using the utilities calculated during the policy evaluation step by taking the action that maximizes the expected utility. These 2 steps are alternated and the algorithm stops once the current policy is stable.

The policy evaluation step uses the following Bellman Update:

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

The policy improvement step uses the following equation:

$$\pi^{i+1}(s) = \arg\max_{a \in A(s)} \sum_{s'} P(s' | s, a) U^{\pi_i}(s')$$

5.1. Pseudo-code for Policy Iteration

The Pseudo-code from “Artificial Intelligence: A Modern Approach” by S. Russell and P. Norvig. Prentice-Hall, third edition, 2010 was used to implement the algorithm.

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**
        $\pi[s] \leftarrow \operatorname{argmax}_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\ U[s']$
        $unchanged? \leftarrow$ false
  until $unchanged?$
  return π

5.2. Implementation

The Policy Iteration algorithm is implemented in policy_iteration.py. When test_policy_iteration.py is run, the following steps occur:

1. The environment (grid world) is created
2. Constants for Policy Iteration algorithm are initialized
3. The MDP is solved using Policy Iteration
4. A plot of the optimal policy and final utilities is displayed
5. The data required for analysis is saved as a .csv file

A set of constants are fixed in test_value_iteration.py

GAMMA (Discount Factor)	0.99
K	100

The value of ‘k’ represents the number of iterations of Bellman updates in the policy evaluation step. A large value of k can slow down the algorithm but it has a good chance to find the optimal policy. A small value of k can speed up the algorithm but it may not find the optimal policy. In this assignment, the value of k is set to 100.

5.2.1. Initial Policy

The starting policy can be any policy. In this case, the initial policy is set to 'UP' for every state. The choice of starting policy affects the number of iterations. The algorithm initializes the starting policy in the *get_starting_policy* method.

```
policy = [[(-1, 0) for _ in range(mdp.grid_width)] for _ in range(mdp.grid_height)]
return policy
```

Initialization of starting policy

5.2.2. Solve method

The MDP is solved using the *solve* method of the PolicyIteration class. The method takes an MDP as an argument and performs a series of policy evaluation and policy improvement steps until the policy is stable. The *policy_evaluation* method returns the new utility values and this is passed as a parameter to the *policy_improvement* method. The *policy_improvement* method returns the new policy and a boolean to determine if the policy is stable. The policy is said to be stable if the new policy is the same as the old policy.

```
def solve(self, mdp):

    # Initialize the staring policy
    policy = self.get_starting_policy(mdp)

    # Initialize the starting utilities
    utilities = np.zeros((mdp.grid_height, mdp.grid_width), dtype=np.float)

    # Initialize the analysis data
    for i in range(utilities.shape[0]):
        for j in range(utilities.shape[1]):
            cur_state = (j, i)
            self.data[f'{cur_state}'] = [0]

    # Initialize the total_iterations
    total_iterations = 0

    # Loop control variable
    is_policy_stable = False

    # Loop while the policy is not stable
    while not is_policy_stable:
        # Policy Evaluation
        utilities, iterations = self.policy_evaluation(policy, utilities, mdp)
        total_iterations += iterations

        # Policy Improvement
        policy, is_policy_stable = self.policy_improvement(policy, utilities, mdp)

    # Return utilities, policy and iterations as a dictionary
    return {"utilities": utilities, "policy": policy, "iterations": total_iterations}
```

Policy Iteration algorithm

5.2.3. Policy Evaluation

The policy evaluation step is performed in the *policy_evaluation* method of the PolicyIteration Class. This method takes the current policy, current utilities and the MDP as arguments and performs updates to the utilities of every state using the current policy. The method loops through all the states and updates the utility of every state by taking the action according to its policy. The algorithm performs the update 'k' times and returns the new utilities. In this assignment the value of 'k' is set to 100.

Below is the code snippet explaining the Bellman update performed in policy evaluation.

```
            # Get action from current policy
            action = policy[i][j]

            # Get transition model of the MDP
            transition_model = mdp.transition_model(cur_state, action)

            action_value = 0

            # Loop through all the next states in the transition model
            # This loop calculates expected utility for taking the action
            for next_state in transition_model:
                utility = utilities[next_state[0]][next_state[1]]
                probability = transition_model[next_state]
                expected_value = probability * utility
                action_value += expected_value

            # Reward of current state
            reward = mdp.receive_reward(cur_state)

            # Bellman Update
            utility = reward + self.gamma * action_value
            new_utilities[i][j] = utility

            # Update analysis data
            self.data[f'{state_format}'].append(utility)

    utilities = new_utilities.copy()

return utilities, iteration
```

Policy Improvement using Bellman update

5.2.4. Policy Improvement

The policy improvement step is performed in the policy_improvement method of the PolicyIteration Class. This method takes the current policy, current utilities and the MDP as arguments and performs updates to the policy using the current utilities. The algorithm updates the policy by taking the greedy action (action that maximizes expected utility) and returns the new policy and a boolean depicting whether the policy

is stable. The policy is said to be stable when the old policy is the same as the new policy. Once the policy is stable, the policy iteration algorithm stops and returns the final policy and utilities.

Below is the code snippet that explains the policy evaluation algorithm. The first explains the policy update step and the second snippet explains the logic to check is the policy is stable

```
# Get all possible actions
actions = mdp.actions
action_values = {}

# Loop through all possible actions
# This loop calculates action values for all actions
for action in actions:
    action_value = 0

    # Get transition model
    transition_model = mdp.transition_model(cur_state, action)

    # Loop through all the next states in the transition model
    # This loop calculates expected utility for taking the action
    for next_state in transition_model:
        utility = utilities[next_state[0]][next_state[1]]
        probability = transition_model[next_state]
        expected_utility = probability * utility
        action_value += expected_utility

    # Set action value for the action
    action_values[action] = action_value

best_action = None
best_action_value = -math.inf

# This loop chooses the action that maximizes expected utility
for action in action_values:
    if action_values[action] > best_action_value:
        best_action = action
        best_action_value = action_values[action]
new_policy[i][j] = best_action
```

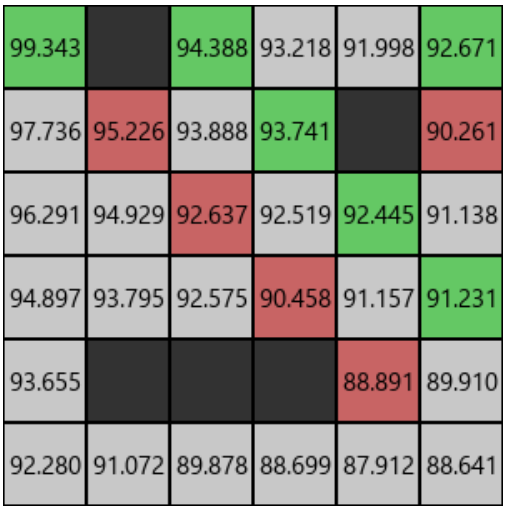
Policy Update by selecting Geedy Action

```
# Checks if the old policy is same as the new policy
# If the old policy is same as the new policy, the policy is stable
for i in range(len(policy)):
    for j in range(len(policy[i])):
        if policy[i][j] != new_policy[i][j]:
            return new_policy, False

return new_policy, True
```

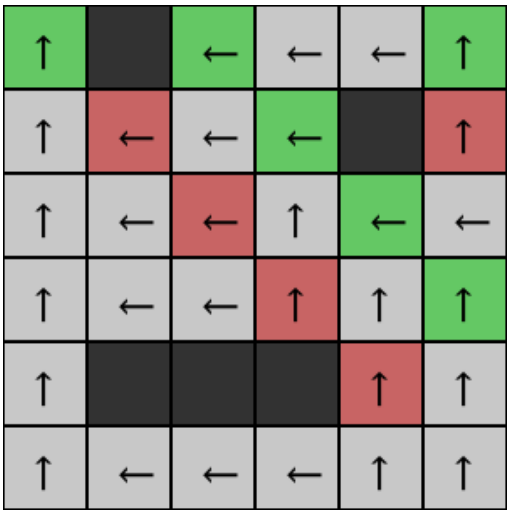
Check is Policy is Stable

5.3. Final Utilities and Optimal Policy

Plots of the final utilities and Optimal Policy are shown below.



Plot of Final Utilities



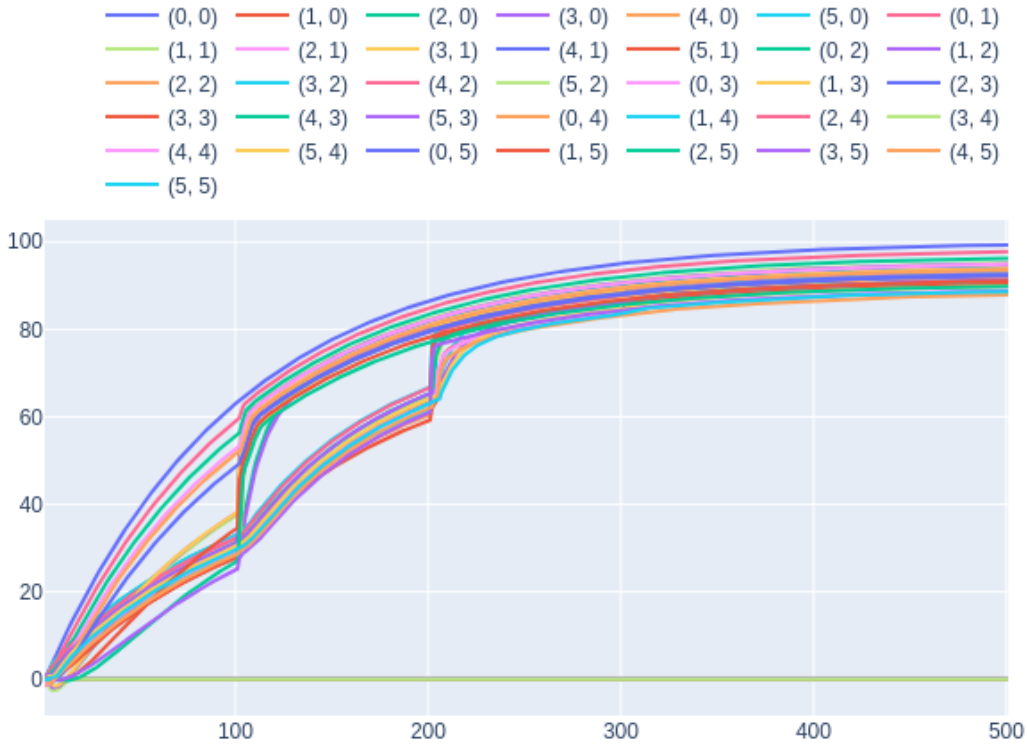
Plot of Optimal Policy

The final utilities when k = 100 are shown. Every state is represented as (col, row).
Number of iterations: 500
(0, 0): 99.34295169575849
(1, 0): 97.73631320642609
(2, 0): 96.29145187761563
(3, 0): 94.89679079725641
(4, 0): 93.65547110771155
(5, 0): 92.28042601275303
(0, 1): 0.0
(1, 1): 95.2259690807233
(2, 1): 94.92937944731963
(3, 1): 93.79544549796816
(4, 1): 0.0
(5, 1): 91.07172932550871
(0, 2): 94.38840892990764
(1, 2): 93.88795006467016
(2, 2): 92.63737931029154
(3, 2): 92.57549711772167
(4, 2): 0.0
(5, 2): 89.87810366927741
(0, 3): 93.21795269266882
(1, 3): 93.74066653181987

(2, 3): 92.5192247178716
(3, 3): 90.45820822627492
(4, 3): 0.0
(5, 3): 88.69936112596669
(0, 4): 91.99756614657267
(1, 4): 0.0
(2, 4): 92.44532076761429
(3, 4): 91.15735876877382
(4, 4): 88.89136479593525
(5, 4): 87.91205077293387
(0, 5): 92.67145472899253
(1, 5): 90.2608748906614
(2, 5): 91.13782276854705
(3, 5): 91.23103625721895
(4, 5): 89.90971736678283
(5, 5): 88.64064228409407

5.4. Plot of Estimated Utilities vs Iterations

A plot of Utility estimates as a function of iterations is shown below



Plot of utility estimates vs iterations

From the above plots, it is clear that the algorithm found the correct policy which is to reach the state (0, 0) and receive infinite reward. The plot of utility estimates vs iterations shows that the policy changes after every 100 iterations and there is a sharp point as positions where iterations = {100, 200}. After iteration 300, the graph suggests that the policy doesn't change by much and it is slowly becoming stable. This could suggest that the smoothness of the curve after every policy improvement step could suggest that the policy is becoming more stable.

5.5. Calculation of best value for K

In order to reduce the number of Bellman updates required to find the optimal policy, the value of K was varied until the algorithm could not find the optimal policy

Results:

K	Iterations	Optimal Policy
100	500	Yes
75	375	Yes
50	250	Yes
10	60	Yes
5	25	No

Note

The above results may differ and will depend on the initial policy chosen. When the starting policy was to take the action 'DOWN' for every state, the results differed and the lowest value of 'k' was 13. The number of Bellman updates also increased and this suggests that the choice of starting policy affects the algorithm.

Plots:

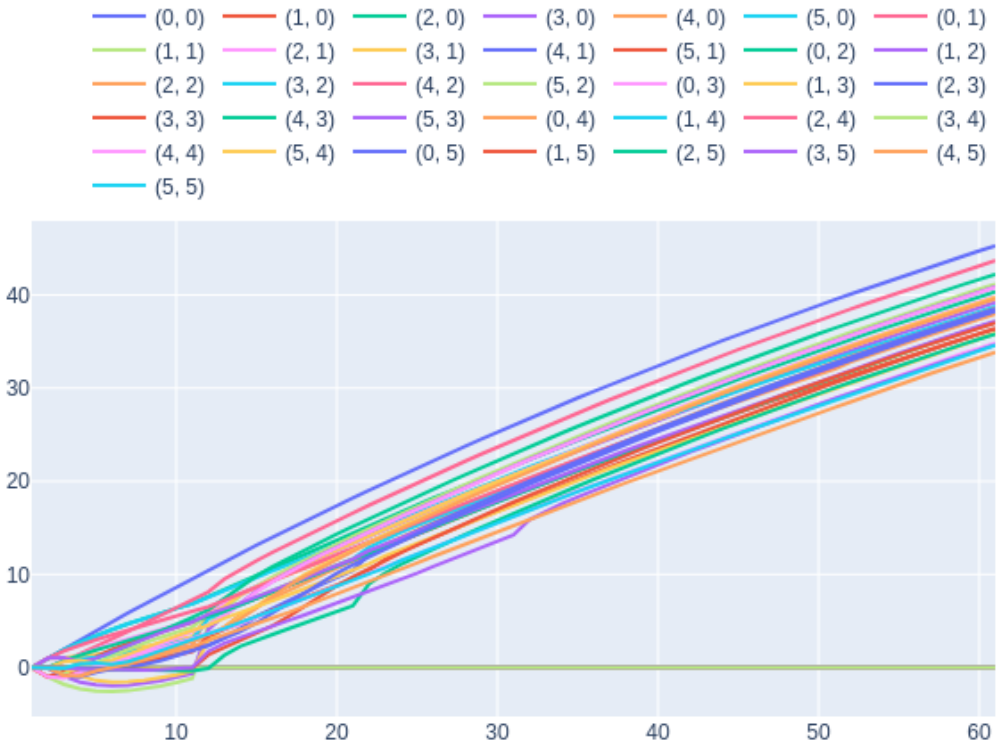
Here are the corresponding plots of optimal policy and final utilities with k = 10

45.284		40.358	39.188	37.971	38.749
43.678	41.167	39.832	39.689		36.352
42.233	40.871	38.579	38.467	38.394	37.104
40.838	39.737	38.517	36.406	37.108	37.197
39.597				34.844	35.875
38.222	37.013	35.819	34.641	33.864	34.603

Plot of Final Utilities

↑		←	←	←	↑
↑	←	←	←		↑
↑	←	←	↑	←	←
↑	←	←	↑	↑	↑
↑				↑	↑
↑	←	←	←	↑	↑

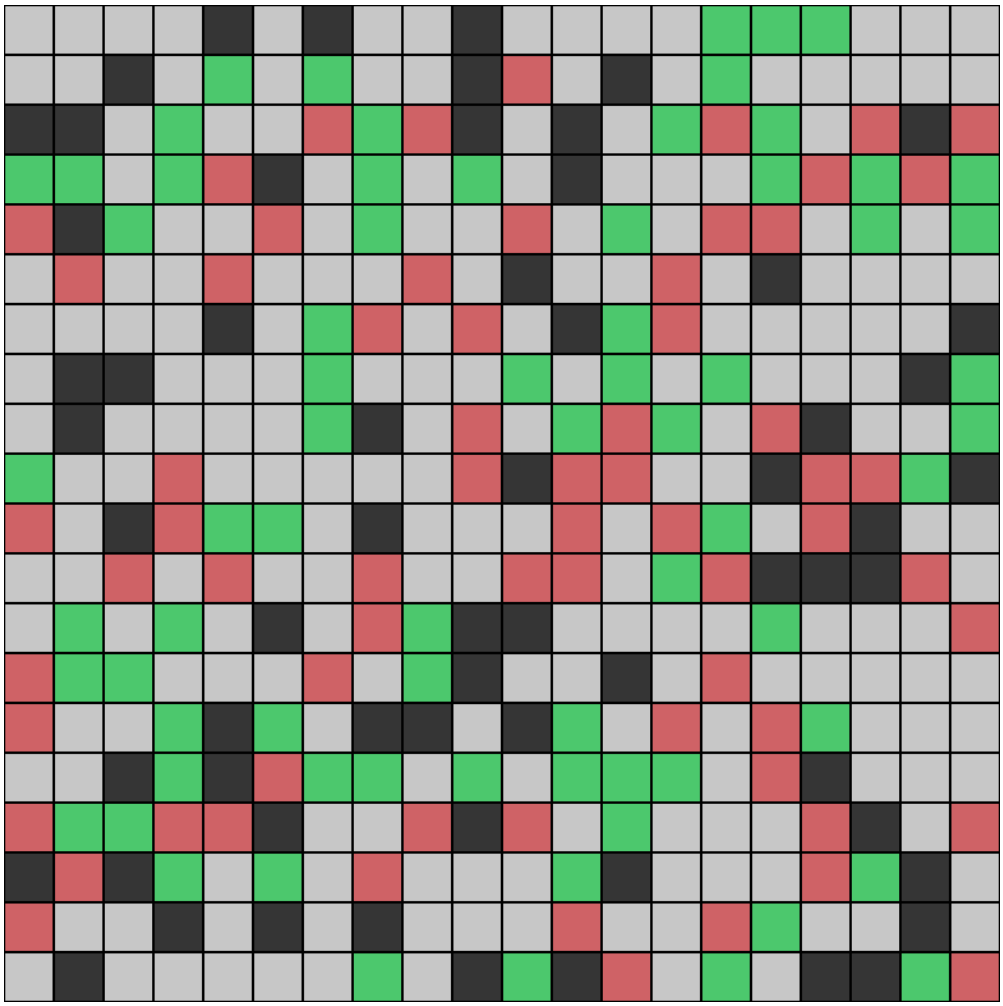
Plot of Optimal Policy



Plot of utility estimates vs iterations

6. Complex Environment

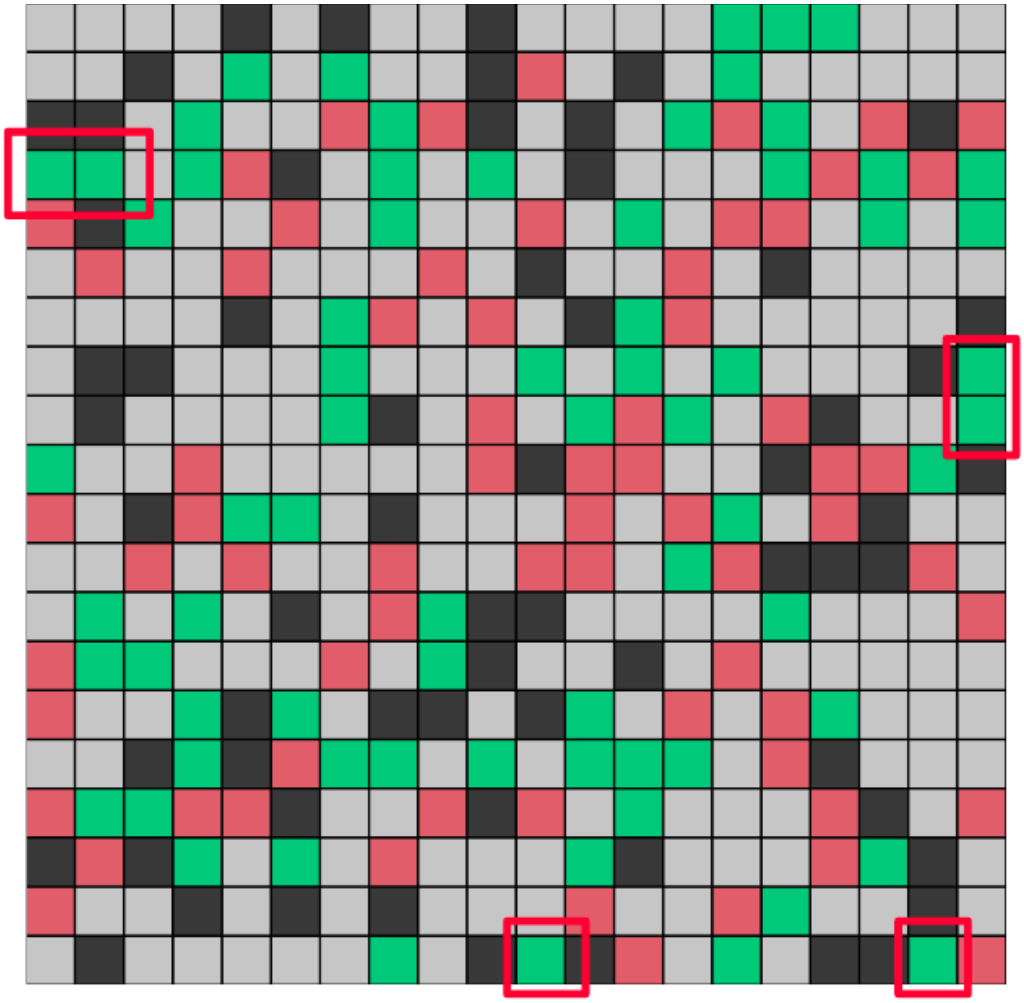
A more complex maze environment was designed to test the value iteration and policy iteration algorithm algorithms. The complex environment can be found in *custom_grid.py*. The size of the grid is 20x20 and the transition probabilities and rewards are the same. Increasing the dimensions of the grid results in more states and this implies longer calculation time and more memory required for the 2 algorithms. The maze environment designed is shown below.



Complex Maze Environment

6.1. Observations

The maze has states where the agent can receive infinite reward. Some of the states are marked below

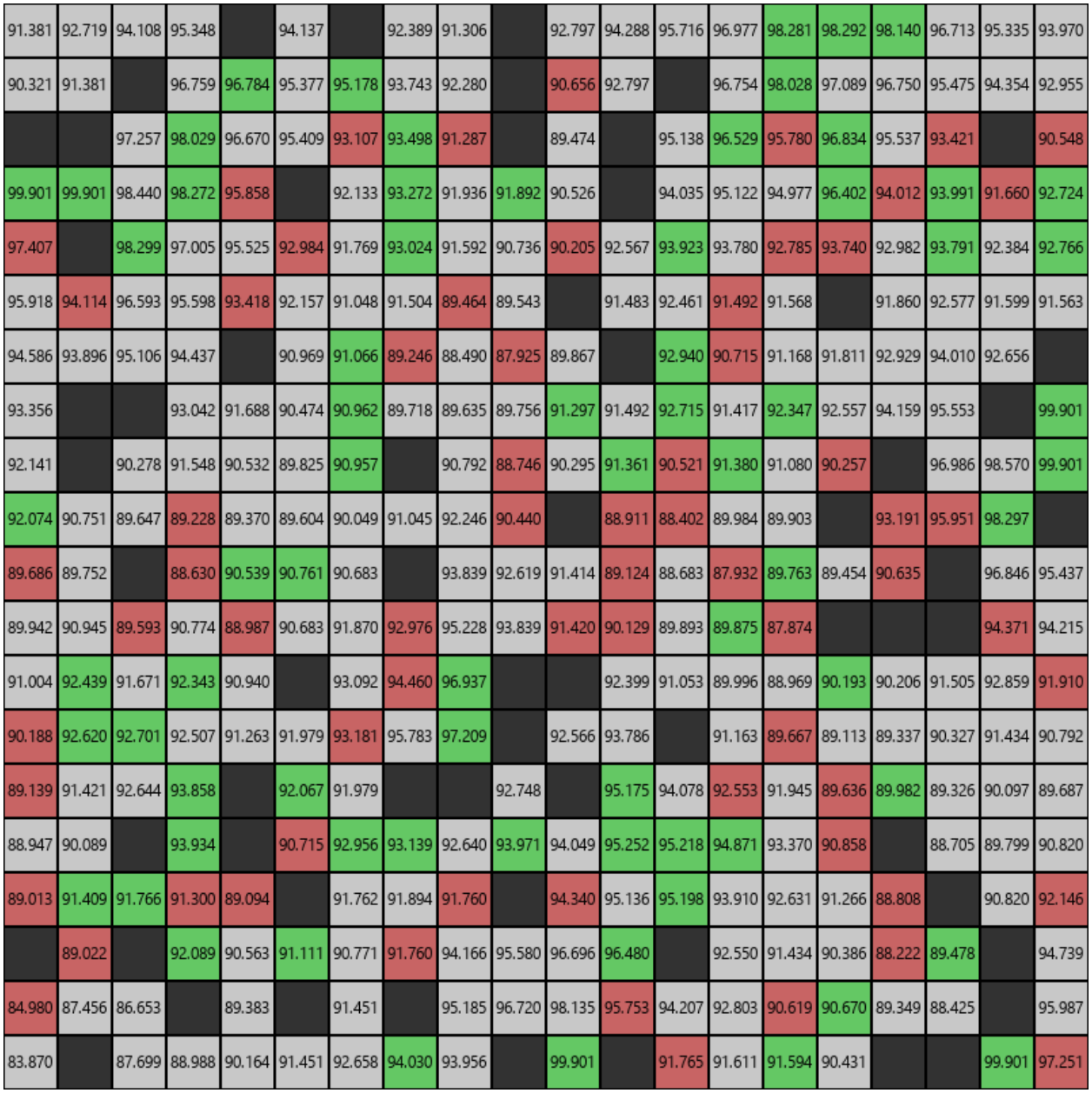


Expected Best States

In the states highlighted above, the agent can either take an action that results in the agent staying in the same state and receiving rewards over and over again or it can take actions that toggle between such states.

6.2. Value Iteration Final Value

The final utilities calculated by the value iteration algorithm are shown below.

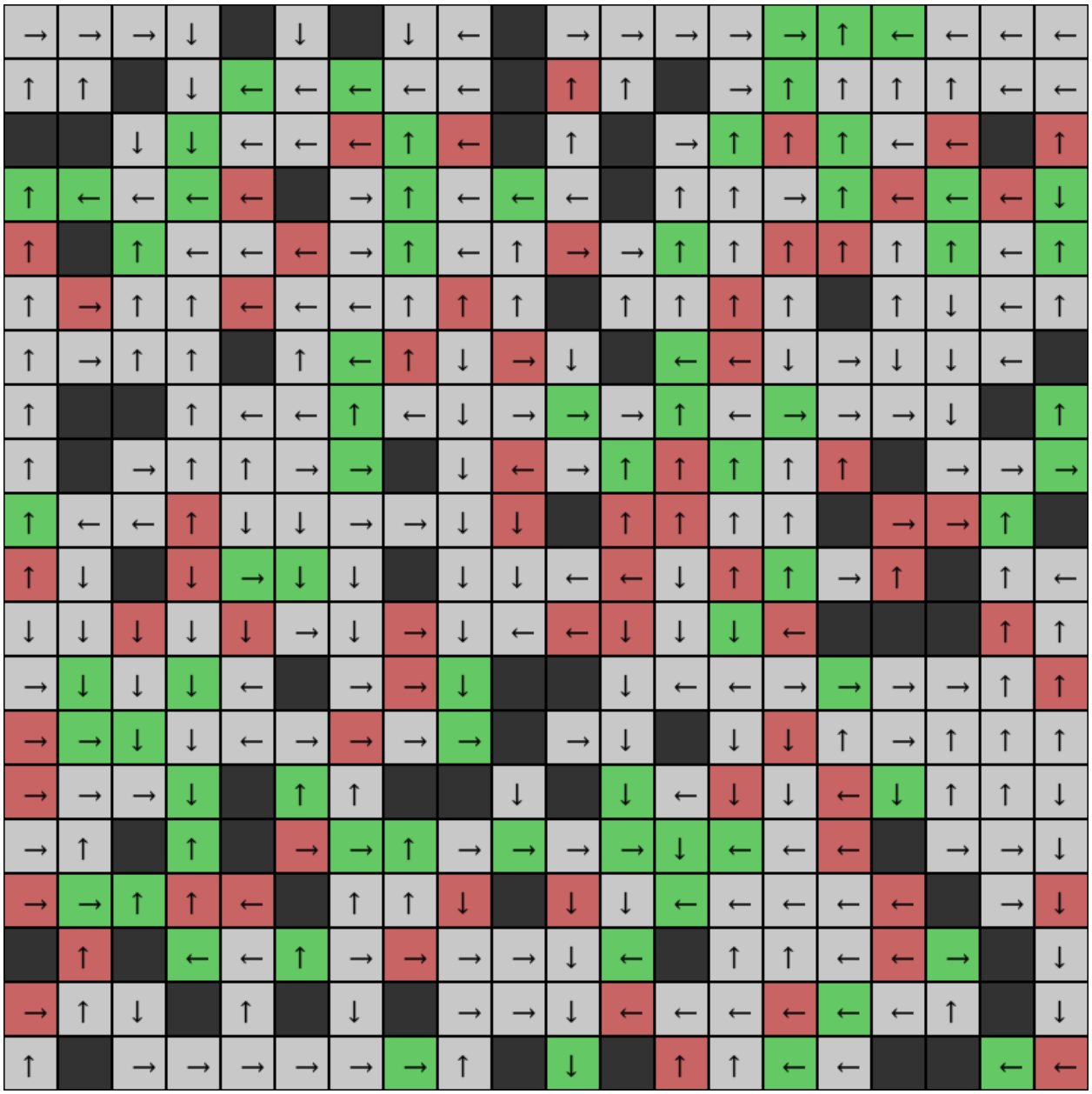


Plot of Final Utilities

The values of the states highlighted earlier are the highest. All of these states have a utility of 99.901 and this could be an indication that the agent found the right policy.

6.3. Value Iteration Policy

The final policy calculated by the value iteration algorithm is shown below.

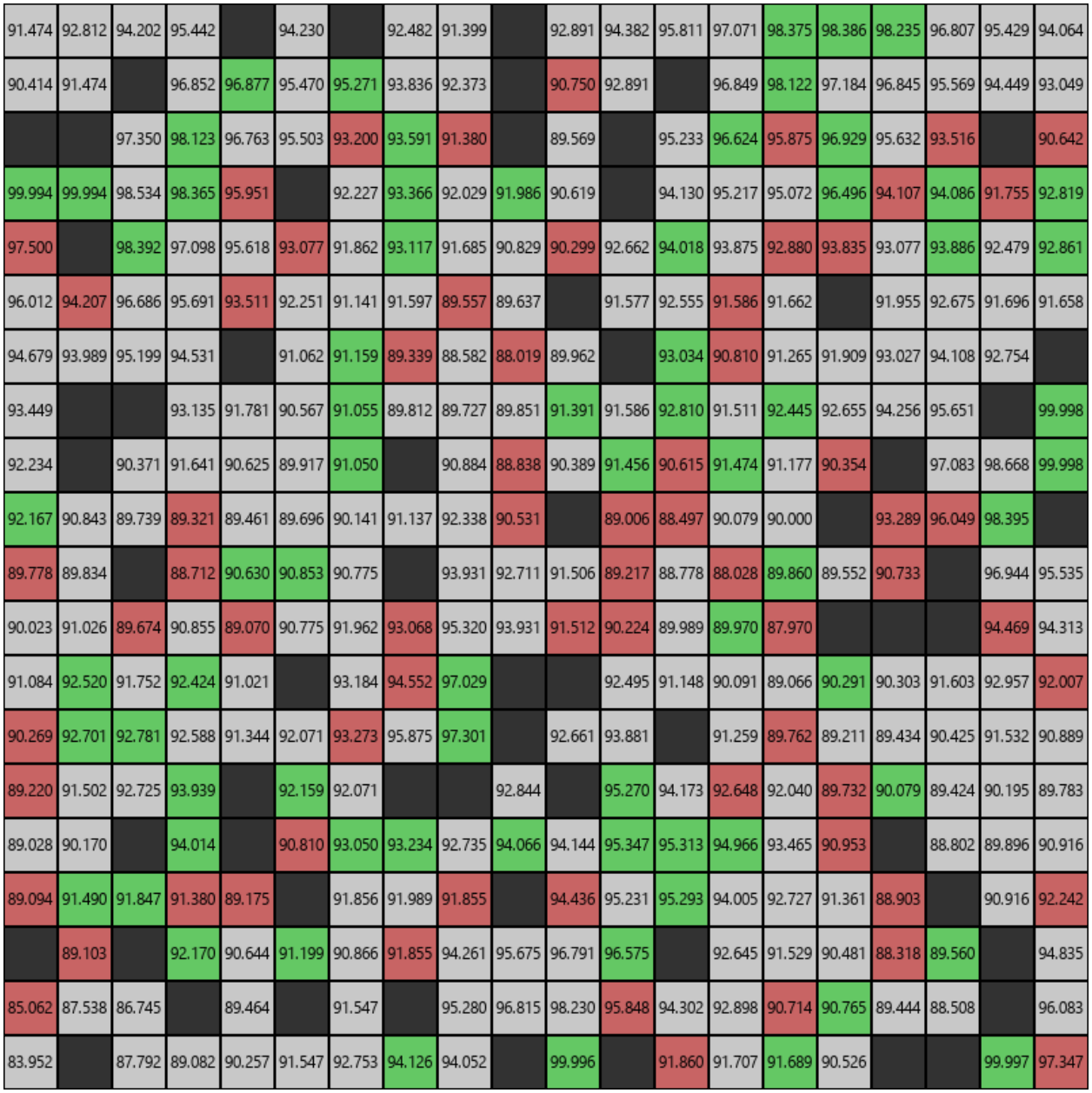


Plot of Optimal Policy

The algorithm finds the optimal policy which is to reach the previously mentioned states. However, the optimal policy can vary because for state (19, 7) ((col, row) format) all four actions will result in the same expected utility. Therefore, the optimal policy can vary.

6.4. Policy Iteration Final Value

The final utilities calculated by the policy iteration algorithm are shown below.

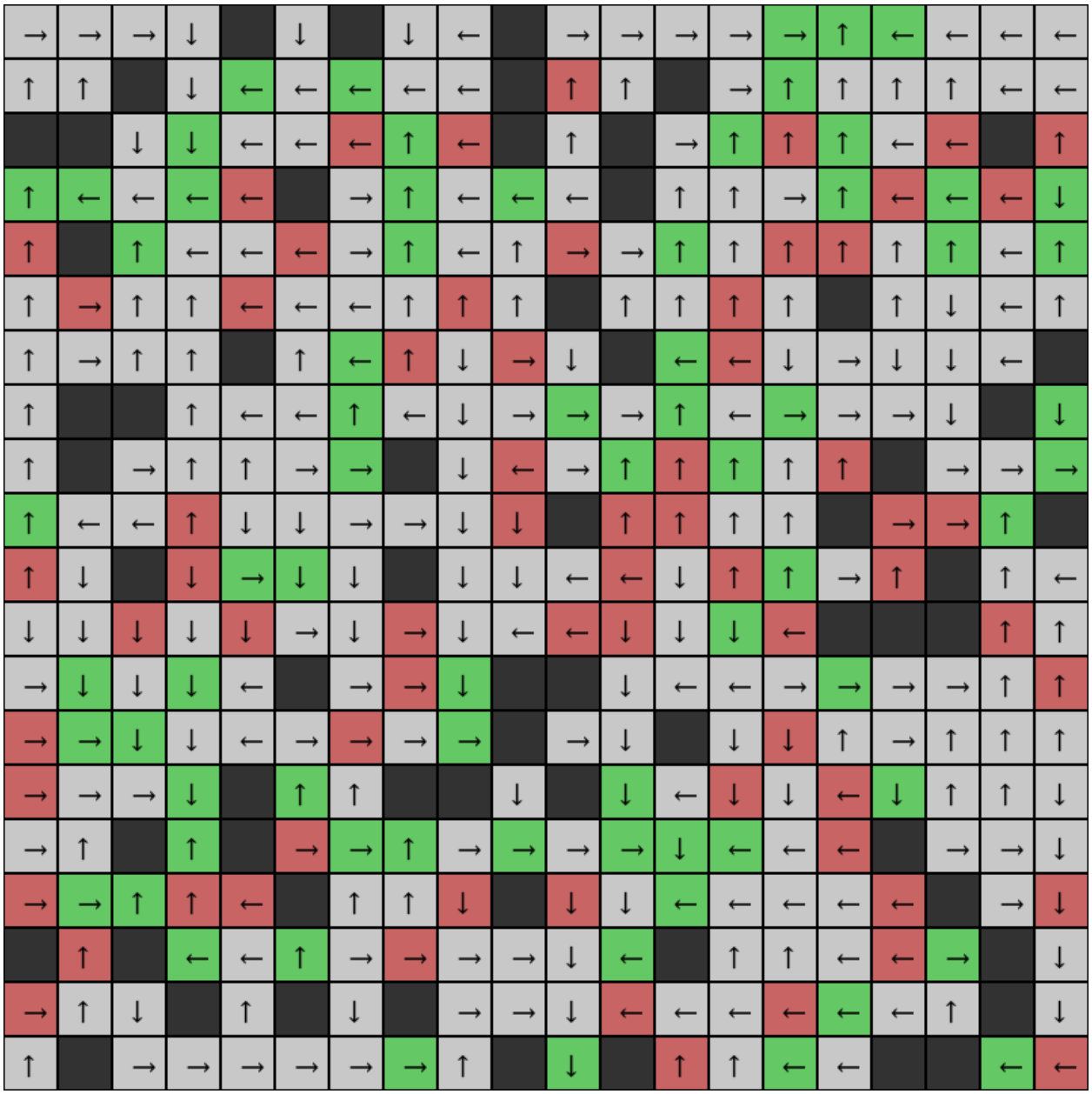


Plot of Final Utilities

The values of the states highlighted earlier are the highest. All of these states have a utility of 99.998 and this could be an indication that the agent found the right policy.

6.5. Policy Iteration Policy

The final policy calculated by the policy iteration algorithm is shown below.



Plot of Optimal Policy

The algorithm finds the optimal policy which is to reach the previously mentioned states. However, the optimal policy can vary because for state (19, 7) ((col, row) format) all four actions will result in the same expected utility. Therefore, the optimal policy can vary.

6.6. Impact of number of states affecting convergence

As the number of states increases, the time and space required for the 2 algorithms also increases. However, the discounted reward equation always converges as long as the discount factor (GAMMA) is less than 1. If gamma was equal to 1, then the equation would not converge since the limit of gamma tending to infinity would not be 0.

The algorithms will converge but it may take a longer time. Not that the choice of programming language also affects the time taken to run the algorithm.

$$U([s_0, s_1, s_2, \ldots]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \ldots$$
$$= \sum_{t=0}^{\infty} \gamma^t R(s_t) \leq \frac{R_{\max}}{1-\gamma} \qquad (0 < \gamma < 1)$$

Discounted Reward

6.7. Impact of maze complexity affecting policy

Both policy and value iteration algorithms are guaranteed to converge to an optimal policy for discounted finite MDPs. Since increasing the number of states does not change the fact that the MDP is finite, the algorithms will still converge for this question. If the maze was to be made in such a way that it had an infinite state space and infinite action space, then the MDP would not guarantee to converge. Since the utilities calculated by these 2 algorithms will converge, the policy learnt by the agent will also be optimal (greedy action).