School of Computer Science and Engineering

CE4046: Intelligent Agents

Assignment 1

GOEL TEJAS
TEJAS005@e.ntu.edu.sg
U1923301G

March 4, 2023

Table of Contents

1. File Structure and Implementation

1.1 Brief Explanation of the Source Code Files

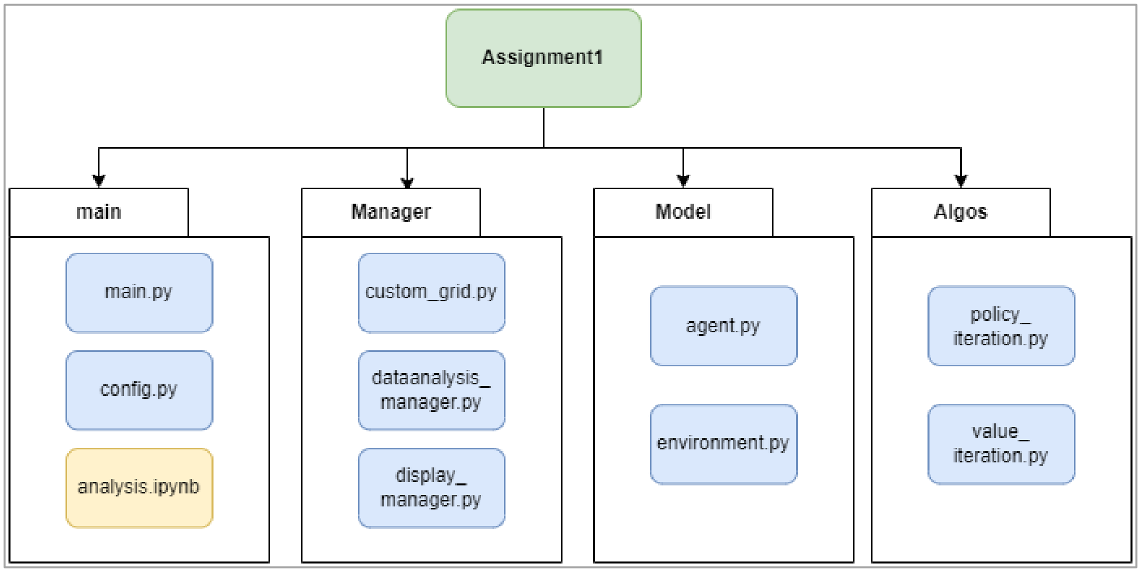


Figure 1.1: File structure of the source code

This assignment requires us to implement Value Iteration and Policy Iteration algorithms to find optimal policies for the grid environment. The assignment has been done using Python and the hierarchy of the implementation code is illustrated in the above figure. The source code folder consists of 2 python files, 1 Jupyter notebook (under main), and 3 packages – manger, model, and algos.

The python files are described as follows:

agent.py: Abstract class that initializes Actions and interface for solving the environment.

environment.py: Class that generates an environment using a grid_world array, rewards array, and the initial state. It also contains the state transformer function and step function.

policy_iteration.py: Concrete class that derives from Agent to implement Policy Iteration algorithm.

value_iteration.py: Concrete class that derives from Agent to implement Value Iteration algorithm.

display_manager.py: Responsible for displaying the final utilities and optimal policies in a Graphical UI.

config.py: Contains the hardcoded values used for running the grid world in Assignment 1.

main.py: Initializes and solves an environment using Value Iteration & Policy Iteration.

To analyze data for utilities for each state during each iteration, two files have been implemented:

dataanalysis_manager.py: Generates the csv files containing state utilities for each iteration.

analysis.ipynb: Jupyter notebook for visualizing the data using plots.

2. Implementation of Grid World Environment

2.1 Grid World and Rewards

The grid world is implemented as a 6x6 two-dimensional array (list) containing walls, white boxes and red boxes. A snippet from *config.py* is shown below:

```
# Grid World
grid = [
    ['G','W','G','Wh','Wh', 'G'],
    ['Wh','R','Wh','G','W', 'R'],
    ['Wh','Wh','R','Wh','G', 'Wh'],
    ['Wh','Wh','Wh','R','Wh', 'G'],
    ['Wh','W','W','W','R', 'Wh'],
    ['Wh','Wh','Wh','Wh','Wh', 'Wh'],
]
```

Grid World Representation for Task 1 in Python

In addition, user has option to opt for Random Grid generation using a command line argument. The code for generating random grid world is shown below (*custom_grid.py*). Grid elements are populated randomly based on the probabilities passed as function arguments.

```
def generate_grid(grid_height: int, grid_width: int, prob_green: float = 0.166, prob_red: float = 0.166,
                  prob_wall: float = 0.168, prob_white: float = 0.5) -> Tuple[List[List], List[List]]:
    assert (prob_green + prob_red + prob_wall + prob_white) == 1.0

    grid_things_arr = ['G', 'R', 'W', 'Wh']
    prob_arr        = [prob_green, prob_red, prob_wall, prob_white]
    reward_map = {'G': +1, 'R': -1, 'W': 0, 'Wh': -0.04}

    grid = np.random.choice(grid_things_arr, (grid_height, grid_width), p=prob_arr)
    rewards = [[reward_map[cell] for cell in row] for row in grid]

    return grid, rewards
```

Python code to generate random Grid World using probabilities

Rewards for the above grid world is generated using a mapping and list comprehension. Green boxes have +1 reward, Red boxes have -1 reward, White boxes have -0.04 reward, and Walls have 0 reward.

```
# Rewards
reward_map = {'G': +1, 'R': -1, 'W': 0, 'Wh': -0.04}
rewards = [[reward_map[cell] for cell in row] for row in grid]
```

Python code to generate rewards for grid world environment

2.2 Actions

The agent can take 4 possible actions which are [UP, LEFT, DOWN, RIGHT]. Each action is represented by the delta from the current position (as a tuple). The first element is the delta along vertical direction while the second element is the delta along horizontal direction. Code snippet from config.py is below:

```python
actions = {
    "UP": (-1, 0),
    "DOWN": (1, 0),
    "RIGHT": (0, 1),
    "LEFT": (0, -1)
}
```

Python code to define the possible actions for an agent

2.3 State Transformer Function

The state transformer function returns all possible next states with their probabilities given the agent's current state and action. The intended action occurs with probability 0.8, and with probability 0.1 the agent moves to either right angle of the intended direction. If the next state after an action is a wall, the agent stays in the same position (which also becomes one of the possible next states). This is implemented as follows in *environment.py*:

```python
    def state_transformer(self, state: Tuple, action: Tuple) -> dict:
        """
        Returns each potential next state with probabilities given the agent's current state and action

        Args:
            state (Tuple): Current state
            action (Tuple): Current action

        Returns:
            model (dict): Next states and probabilities
        """
        model: DefaultDict = defaultdict(int)
        possible_actions: List[Tuple] = [action, (action[1], action[0]), (-action[1], -action[0])]
        probability_actions: Tuple = (0.8, 0.1, 0.1)

        for a, prob in zip(possible_actions, probability_actions):
            next_state: Tuple = (state[0]+a[0], state[1]+a[1])
            next_state = next_state if self.is_valid_state(next_state) else state

            model[next_state] += prob

        return dict(model)
```

Python code implementing State transformer from Grid World

3. Implementation of Value Iteration

3.1 Description and Pseudo-code

The basic idea is to calculate the utility of every state in the environment until convergence, and then use these utilities to select the optimal policy that maximizes expected utility.

The utility of each state is updated using the Bellman equation:

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

The algorithm begins with initializing all utilities as 0 which is then updated each iteration using the above Bellman equation. This is done until convergence is achieved – which is defined using the hyper-parameter ε and discount factor δ.

Pseudo-code for the algorithm was taken from the course textbook.

```
function VALUE-ITERATION(mdp, ε) returns a utility function
  inputs: mdp, an MDP with states S, actions A(s), transition model P(s' | 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 ← U'; δ ← 0
     for each state s in S do
        U'[s] ← R(s) + γ max_{a ∈ A(s)} Σ_{s'} P(s' | s, a) U[s']
        if |U'[s] − U[s]| > δ then δ ← |U'[s] − U[s]|
  until δ < ε(1 − γ)/γ
  return U
```

Pseudo-code for Value Iteration algorithm

3.2 Implementation in Python

The algorithm is implemented in *value_iteration.py* and is tested using *main.py*. Whenever user runs an agent with value iteration algorithm, following occurs:

1) The environment and rewards are initialized.
2) An agent is initialized with "Value Iteration" algorithm and hyper-parameters.
3) The Markov Decision Problem is solved by first calculating utilities of states and the optimal policies.
4) Plot of final utilities and optimal policy is displayed in a graphical window.
5) Data related to utilities of each state during each iteration is saved in csv file.

```python
24      def solve_utilities(self, env: Environment) -> Tuple[np.ndarray, int]:
25          utilities: np.ndarray = np.zeros((env.grid_height, env.grid_width), dtype=np.float64)
26
27          threshold: float = self.epsilon * (1 - self.gamma) / self.gamma
28          iterations: int = 0
29          delta: float = float('inf')
30
31          while delta > threshold:
32              new_utilities: np.ndarray = utilities.copy()
33              delta = 0
34              iterations += 1
35
36              # Iterate through every state in environment
37              for i in range(env.grid_height):
38                  for j in range(env.grid_width):
39                      curr_state: Tuple = (i, j)
40
41                      # If wall or invalid state, skip
42                      if not env.is_valid_state(curr_state): continue
43
44                      # Reward for current state
45                      reward: float = env.get_reward(curr_state)
46
47                      # Finding action that maximizes expected utility
48                      action_expected_utility: List = []
49                      for action in self.ACTIONS:
50                          state_transformer: dict = env.state_transformer(curr_state, action=action.value)        # Get state transformer model for current state and action
51                          expected_utility: float = sum([ utilities[next_state[0]][next_state[1]]*prob \
52                              for next_state, prob in state_transformer.items() ])                               # Calculate expected utility over all possible next states
53                          action_expected_utility.append(expected_utility)
54
55                      # Bellman Update
56                      new_utilities[i][j] = reward + self.gamma * max(action_expected_utility)
57
58                      # Update delta
59                      delta = max(delta, abs(new_utilities[i][j] - utilities[i][j]))
60
61                      # Append state utility for analysis
62                      self.data_analysis[str((j,i))].append(new_utilities[i][j])
63
64              utilities = new_utilities.copy()
65
66          return utilities, iterations
```

Python code for implementing Value Iteration

The threshold for convergence criteria is calculated as $threshold = \epsilon \times (1 - \gamma)/\gamma$. γ is the discount factor, initialized by default as 0.99. During each iteration of algorithm, we iterate through every state in the environment and perform Bellman update (as in line 56 of code). We then update delta to find maximum change in utility (line 59). Algorithm stops when delta is lesser than or equal to threshold.

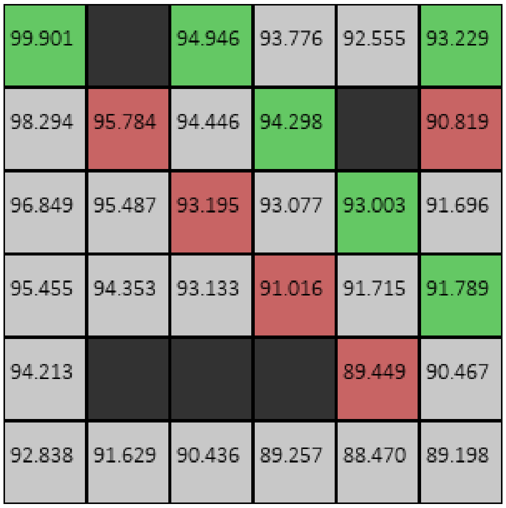
After the utilities of each state are calculated, the optimal policies are selected.

```python
    def solve_optimal_policy(self, utilities: np.ndarray, env: Environment) -> np.ndarray:
        policy = [[None for _ in range(env.grid_width)] for _ in range(env.grid_height)]

        # Iterate through every state in environment
        for i in range(env.grid_height):
            for j in range(env.grid_width):
                curr_state: Tuple = (i, j)

                # If wall or invalid state, skip
                if not env.is_valid_state(curr_state): continue

                # Finding action that maximizes expected utility
                action_expected_utility: dict = {}
                for action in self.ACTIONS:
                    state_transformer: dict = env.state_transformer(curr_state, action=action.value)        # Get state transformer model for current state and action
                    expected_utility: float = sum([ utilities[next_state[0]][next_state[1]]*prob \
                        for next_state, prob in state_transformer.items() ])                               # Calculate expected utility over all possible next states
                    action_expected_utility[action] = expected_utility

                # Update Policy
                policy[i][j] = max(action_expected_utility, key=action_expected_utility.get)

        return policy
```

Calculating optimal policies for each state

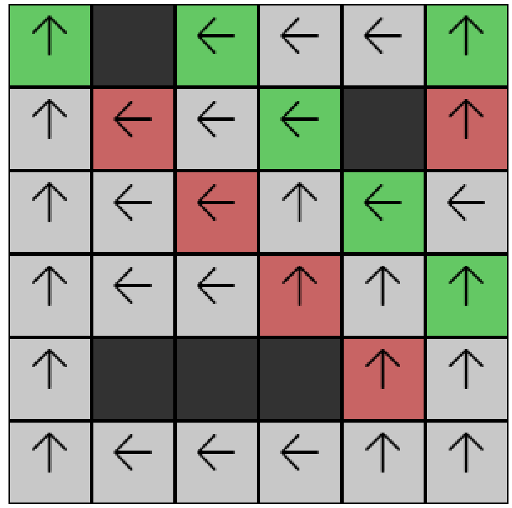
3.3 Final Utilities and Optimal Policy

Plot of final utilities and optimal policy are shown below. These are screenshots of graphical window displayed when running *main.py*.



99.901		94.946	93.776	92.555	93.229
98.294	95.784	94.446	94.298		90.819
96.849	95.487	93.195	93.077	93.003	91.696
95.455	94.353	93.133	91.016	91.715	91.789
94.213				89.449	90.467
92.838	91.629	90.436	89.257	88.470	89.198

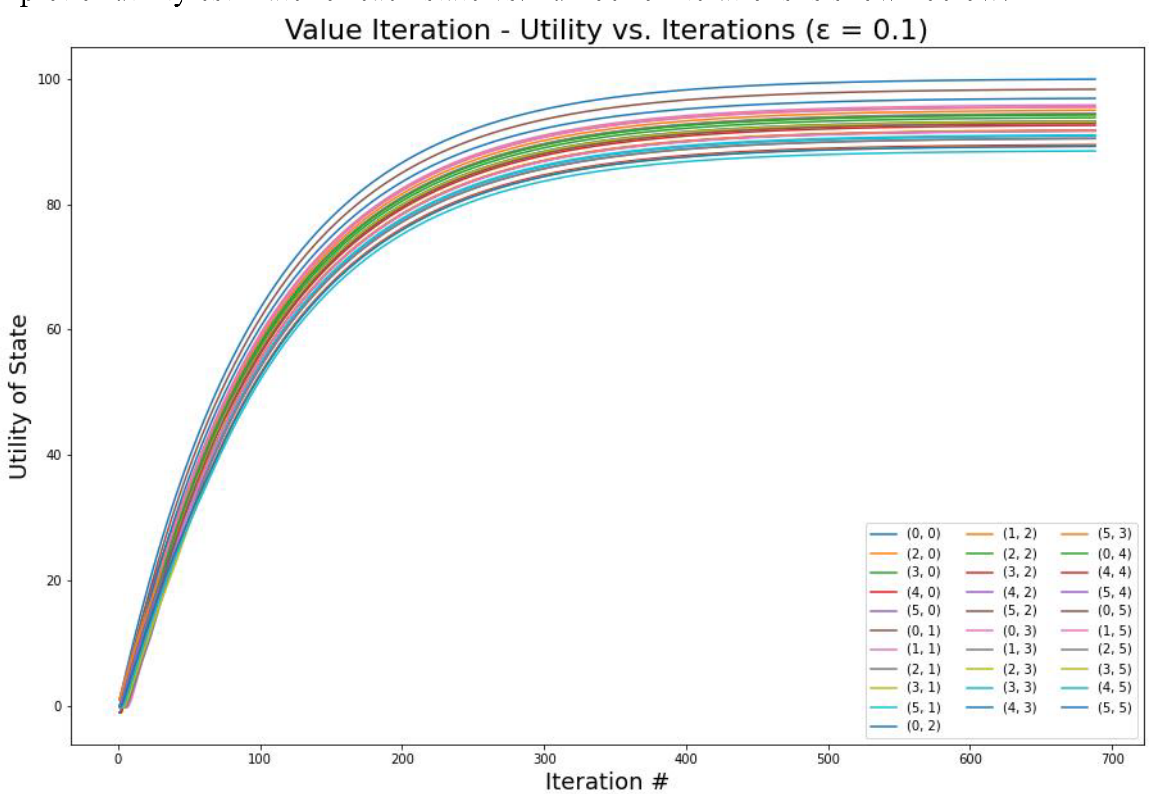
Plot of final utilities for task 1 using Value Iteration



Plot of optimal policies for task 1 using Value Iteration

3.4 Plot of Utility Estimates as a function of the number of iterations

A plot of utility estimate for each state vs. number of iterations is shown below:



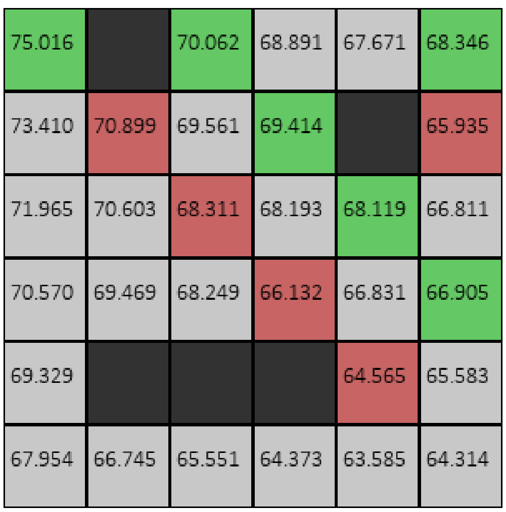
Plot of utility estimates vs. iterations for Value Iteration

3.5 Analysis with different hyper-parameters

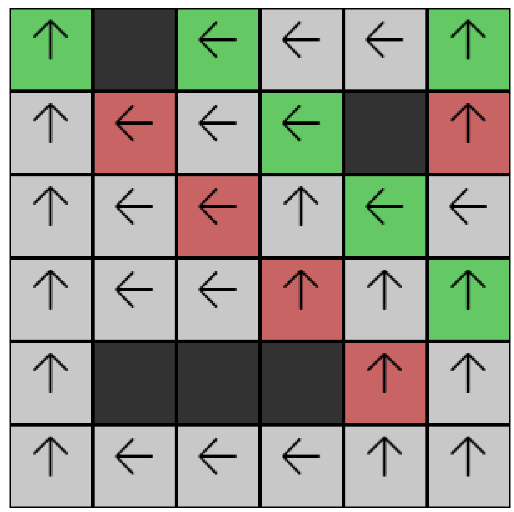
The value of ε was varied to find required optimal policy and recorded as follows:

ε	Iterations	Optimal Policy?
0.1	688	Yes
1	459	Yes
25	138	Yes
45	80	Yes
50	69	No

The utility and optimal policies are shown below for ε = 25:

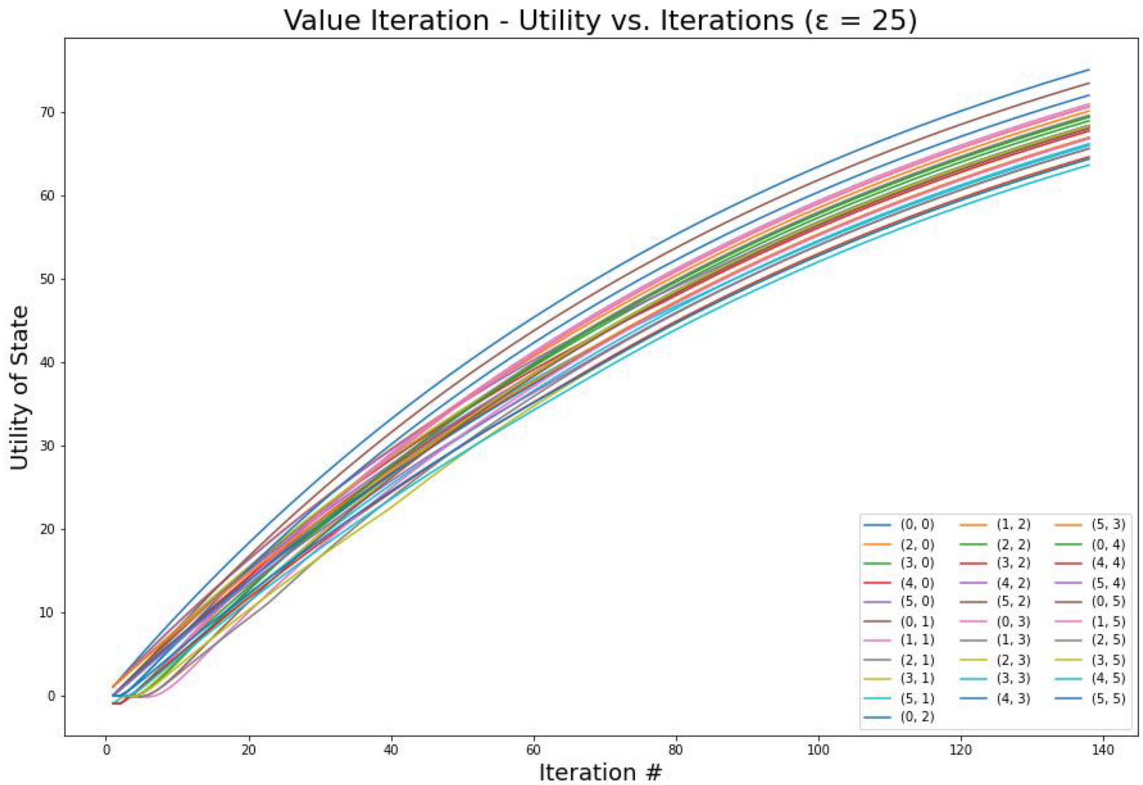


Plot of final utilities for task 1 using Value Iteration



Plot of optimal policies for task 1 using Value Iteration

A plot of utility estimate for each state vs. number of iterations is shown below:



Plot of utility estimates vs. iterations for Value Iteration

4. Implementation of Policy Iteration

4.1 Description and Pseudo-code

Policy iteration relies on the idea that the exact magnitude of the utilities of the states need not be precise to get the optimal policy. Policy iteration alternates between two steps, beginning from some initial policy π_0.

- Policy Evaluation: Given a policy π_i, calculate U_i, utility of every state if π_i were to be executed.
- Policy Improvement: Calculate a new policy π_{i+1} using one-step look ahead based on U_i.

Algorithm terminates when the policy improvement step yields no change in utilities. One should note that the policy evaluation step involves a simplified version of the Bellman equation:

$$U_i(s) = R(s) + \gamma \sum_{s'} P(s' \mid s, \pi_i(s)) U_i(s') \; .$$

The above equation can be solved using exact solution methods in $O(n^3)$ time; however, this is efficient only for small state spaces. For larger state spaces, we perform *k* iteration steps to give an approximation of the utilities. The resulting algorithm is more efficient than value iteration or standard policy iteration.

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

Pseudo-code for the algorithm was taken from the course textbook.

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

Pseudo-code for Policy Iteration algorithm

4.2 Implementation in Python

The algorithm is implemented in policy*_iteration.py* and is tested using *main.py*. Whenever user runs an agent with value iteration algorithm, following occurs:

1) The environment and rewards are initialized.
2) An agent is initialized with “Policy Iteration” algorithm and hyper-parameters.
3) The Markov Decision Problem is solved by alternating between the Policy Iteration steps.
4) Plot of final utilities and optimal policy is displayed in a graphical window.
5) Data related to utilities of each state during each iteration is saved in csv file.

```python
    def solve(self, env: Environment) -> dict:
        # Initialize random policy
        policy: List[List] = [[random.choice(list(self.ACTIONS)) for _ in range(env.grid_width)] for _ in range(env.grid_height)]

        # Initialize utilities
        utilities: np.ndarray = np.zeros((env.grid_height, env.grid_width), dtype=np.float64)

        total_iterations: int = 0

        is_policy_unchanged = False
        while not is_policy_unchanged:
            utilities, iterations = self.policy_evaluation(policy, utilities, env)
            total_iterations += iterations

            policy, is_policy_unchanged = self.policy_improvement(policy, utilities, env)

        return {
            "utilities": utilities,
            "policy": policy,
            "iterations": iterations,
            "algorithm": "policy_iteration"
        }
```

Python code for implementing Policy Iteration

The algorithm begins by initializing a policy vector with randomized policy for each state. A utilities vector is also initialized with 0 for all states. The Policy Evaluation and Policy Iteration steps are then repeated until policy is unchanged.

```python
    def policy_evaluation(self, policy: List[List], utilities: np.ndarray, env: Environment) -> Tuple[np.ndarray, int]:
        """...
        iterations: int = 0

        while iterations < self.k:
            new_utilities: np.ndarray = utilities.copy()
            iterations += 1

            # Iterate through every state in environment
            for i in range(env.grid_height):
                for j in range(env.grid_width):
                    curr_state: Tuple = (i, j)

                    # If wall or invalid state, skip
                    if not env.is_valid_state(curr_state): continue

                    # Reward for current state
                    reward: float = env.get_reward(curr_state)

                    # Finding expected utility for the action given by current policy
                    action = policy[i][j]
                    state_transformer: dict = env.state_transformer(curr_state, action=action.value)          # Get state transformer model for current state and action
                    action_expected_utility: float = sum([ utilities[next_state[0]][next_state[1]]*prob \
                            for next_state, prob in state_transformer.items() ])                                # Calculate expected utility over all possible next states

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

                    # Append state utility for analysis
                    self.data_analysis[str((j,i))].append(new_utilities[i][j])

            utilities = new_utilities.copy()

        return utilities, iterations
```

Python code for implementing Policy Evaluation

Policy evaluation iterations are repeated k times (default - 300) to perform Bellman update of utilities of every state. The discount factor γ used in the Bellman equation is initialized as 0.99 be default.

```python
    def policy_improvement(self, policy: List[List], utilities: np.ndarray, env: Environment) -> Tuple[List[List], bool]:
        is_policy_unchanged = True

        # Iterate through every state in environment
        for i in range(env.grid_height):
            for j in range(env.grid_width):
                curr_state: Tuple = (i, j)

                # If wall or invalid state, skip
                if not env.is_valid_state(curr_state): continue

                # Finding action that maximizes expected utility
                action_expected_utility: dict = {}
                for action in self.ACTIONS:
                    state_transformer: dict = env.state_transformer(curr_state, action=action.value)       # Get state transformer model for current state and action
                    expected_utility: float = sum([ utilities[next_state[0]][next_state[1]]*prob \
                        for next_state, prob in state_transformer.items() ])                               # Calculate expected utility over all possible next states
                    action_expected_utility[action] = expected_utility

                # Update Policy
                best_action = max(action_expected_utility, key=action_expected_utility.get)
                if policy[i][j] != best_action:
                    policy[i][j] = best_action
                    is_policy_unchanged = False

        return policy, is_policy_unchanged
```

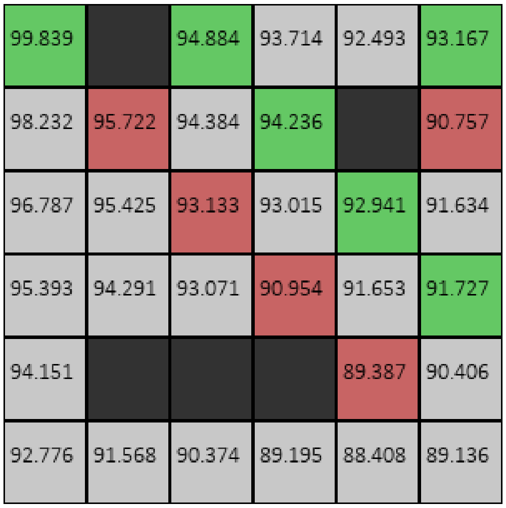
Python code for implementing Policy Improvement

After utilities of states are updated, new policy is calculated for the states by using one-step look ahead to maximize expected utility of the state. If policy of a state changes, a flag is set which indicates that Policy Evaluation step must be repeated.

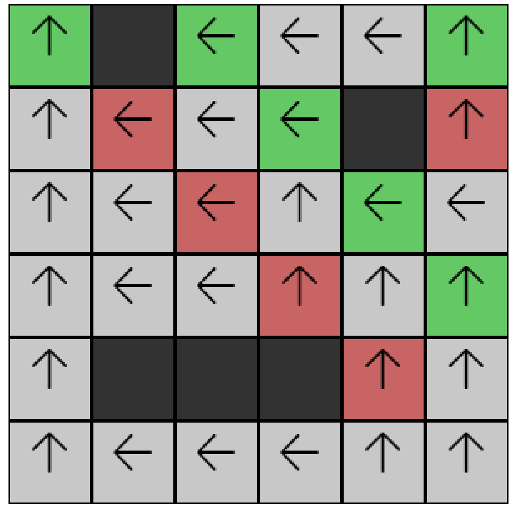
After the algorithm halts, utilities and optimal policies of states are displayed.

4.3 Final Utilities and Optimal Policy

Plot of final utilities and optimal policy are shown below. These are screenshots of graphical window displayed when running *main.py*.



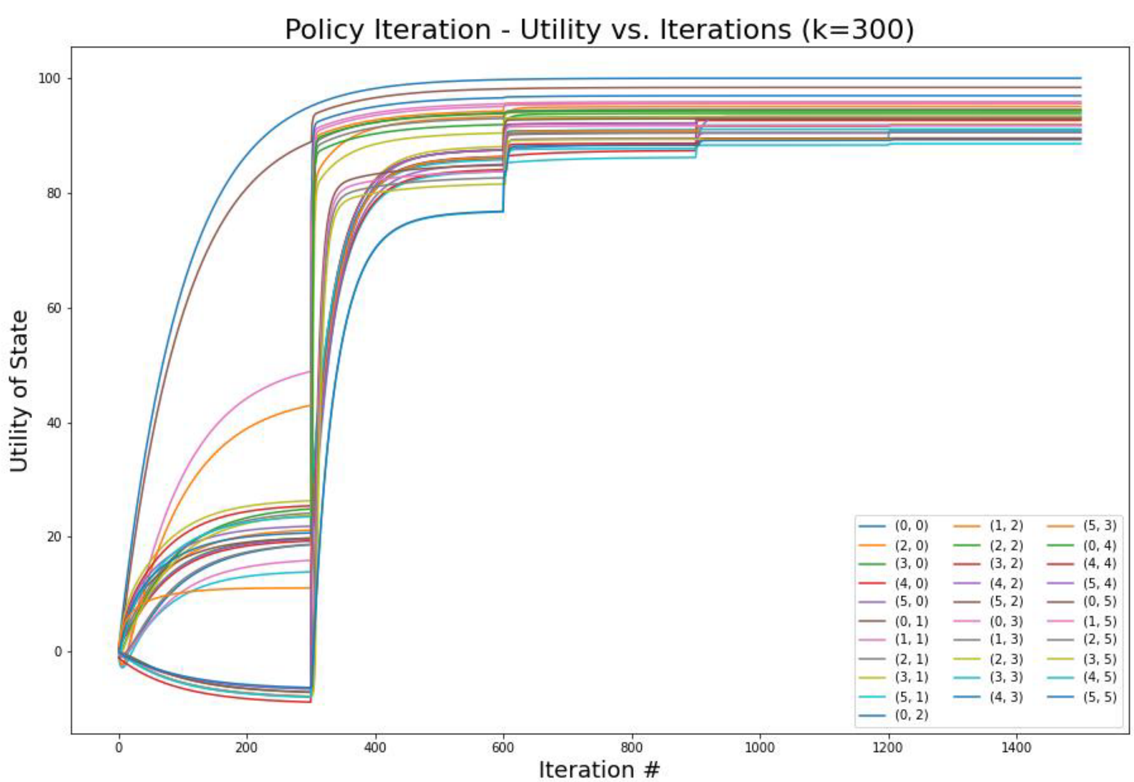
Plot of final utilities for task 1 using Policy Iteration



Plot of optimal policies for task 1 using Policy Iteration

4.4 Plot of Utility Estimates as a function of the number of iterations

A plot of utility estimate for each state vs. number of iterations is shown below:



Plot of utility estimates vs. iterations for Policy Iteration

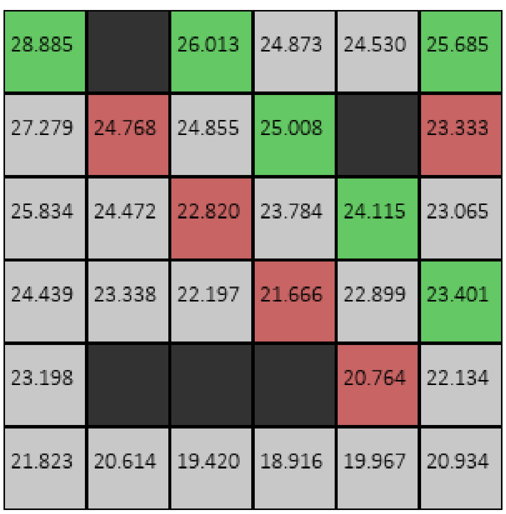
4.5 Analysis with different hyper-parameters

The value of ε was varied to find required optimal policy and recorded as follows:

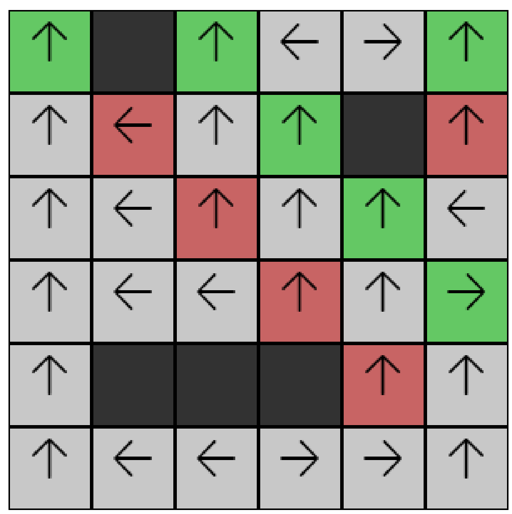
k	Iterations	Optimal Policy?
300	1500	Yes
100	600	Yes
75	525	Yes
50	350	Yes
10	80	Yes
5	35	No

Note: The above results may differ and will depend on the initial policy chosen (randomly chosen in current implementation).

The utility and optimal policies are shown below for k = 5:

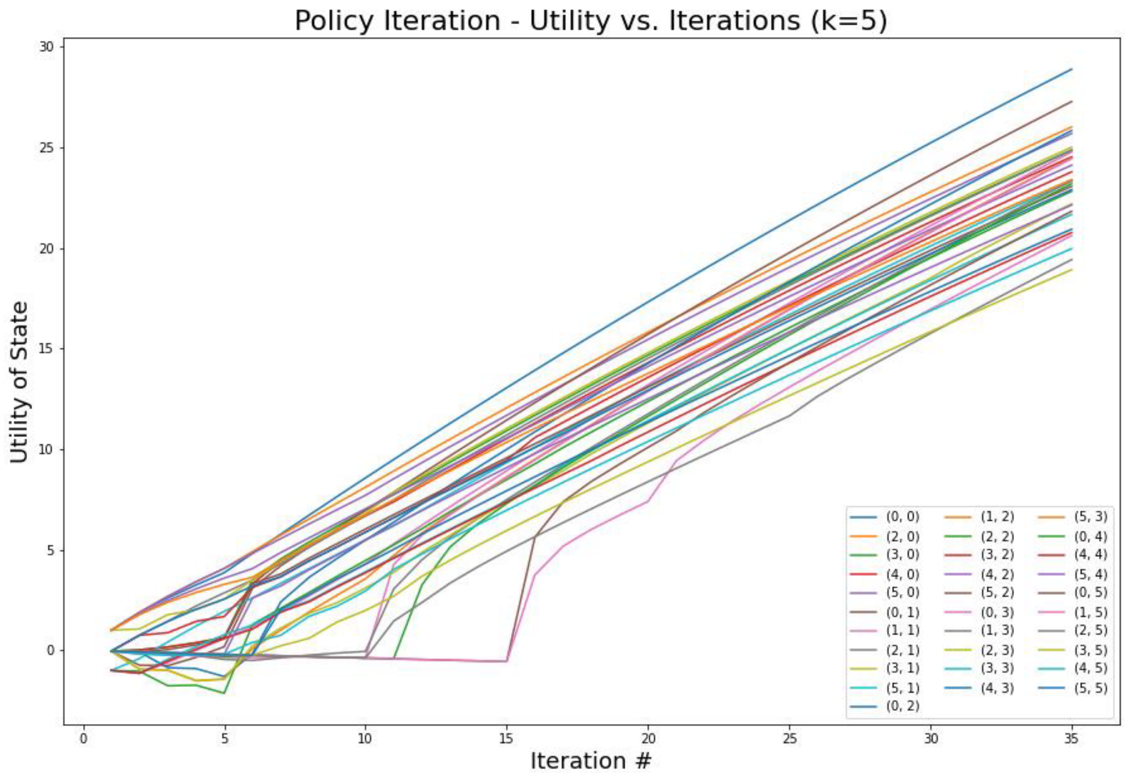


Plot of final utilities for task 1 using Policy Iteration



Plot of optimal policies for task 1 using Policy Iteration

A plot of utility estimate for each state vs. number of iterations is shown below:

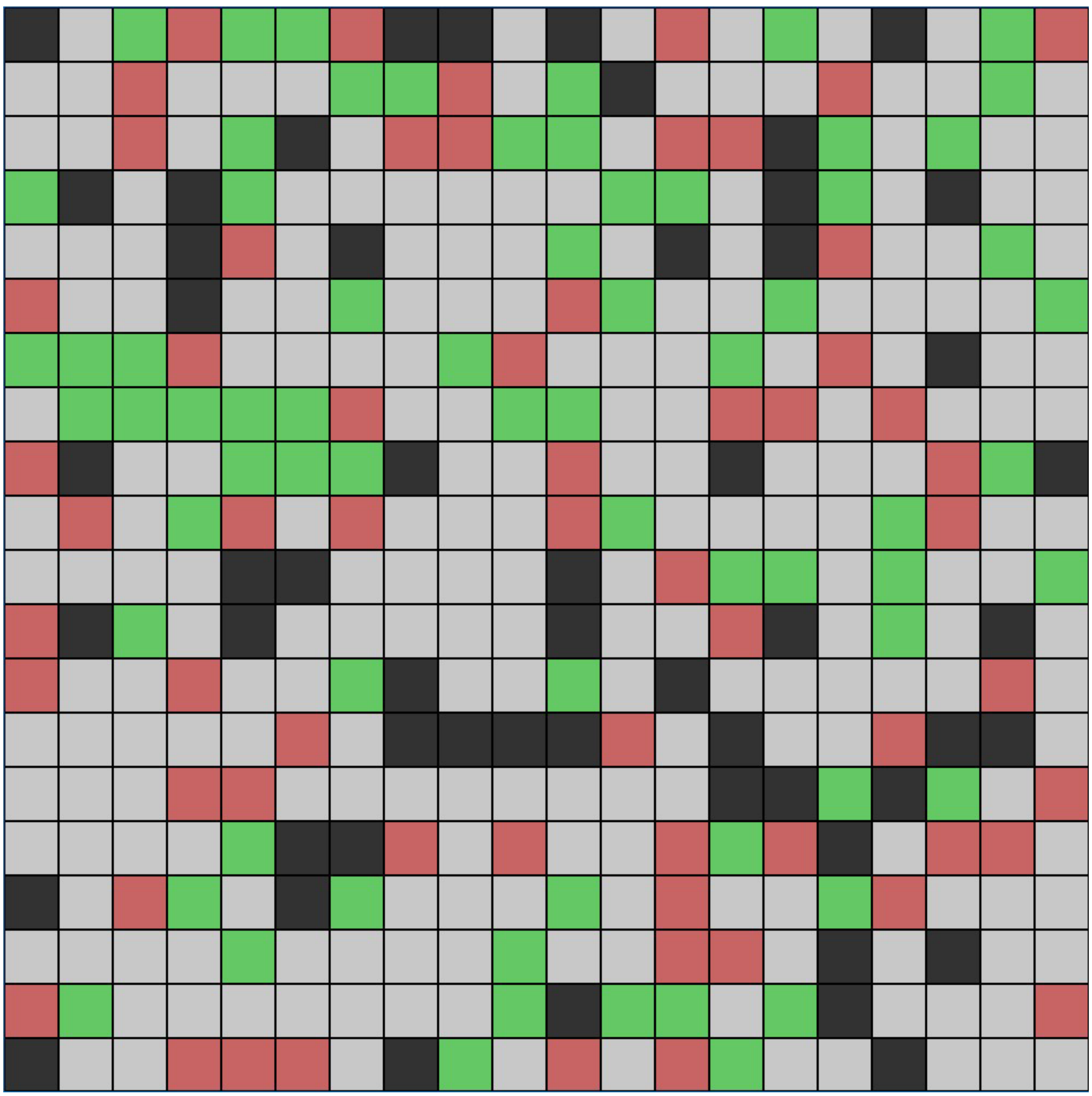


Plot of utility estimates vs. iterations for Policy Iteration

5. Design of More Complicated Maze Environment

In this section, the two algorithms, Value Iteration and modified Policy Iteration are tested by running them in a complex maze environment. In this case, complexity of maze environment is increased by increasing size of the environment. Increasing environment size increases the number of states which results in longer calculation time and larger memory required to run the two algorithms.

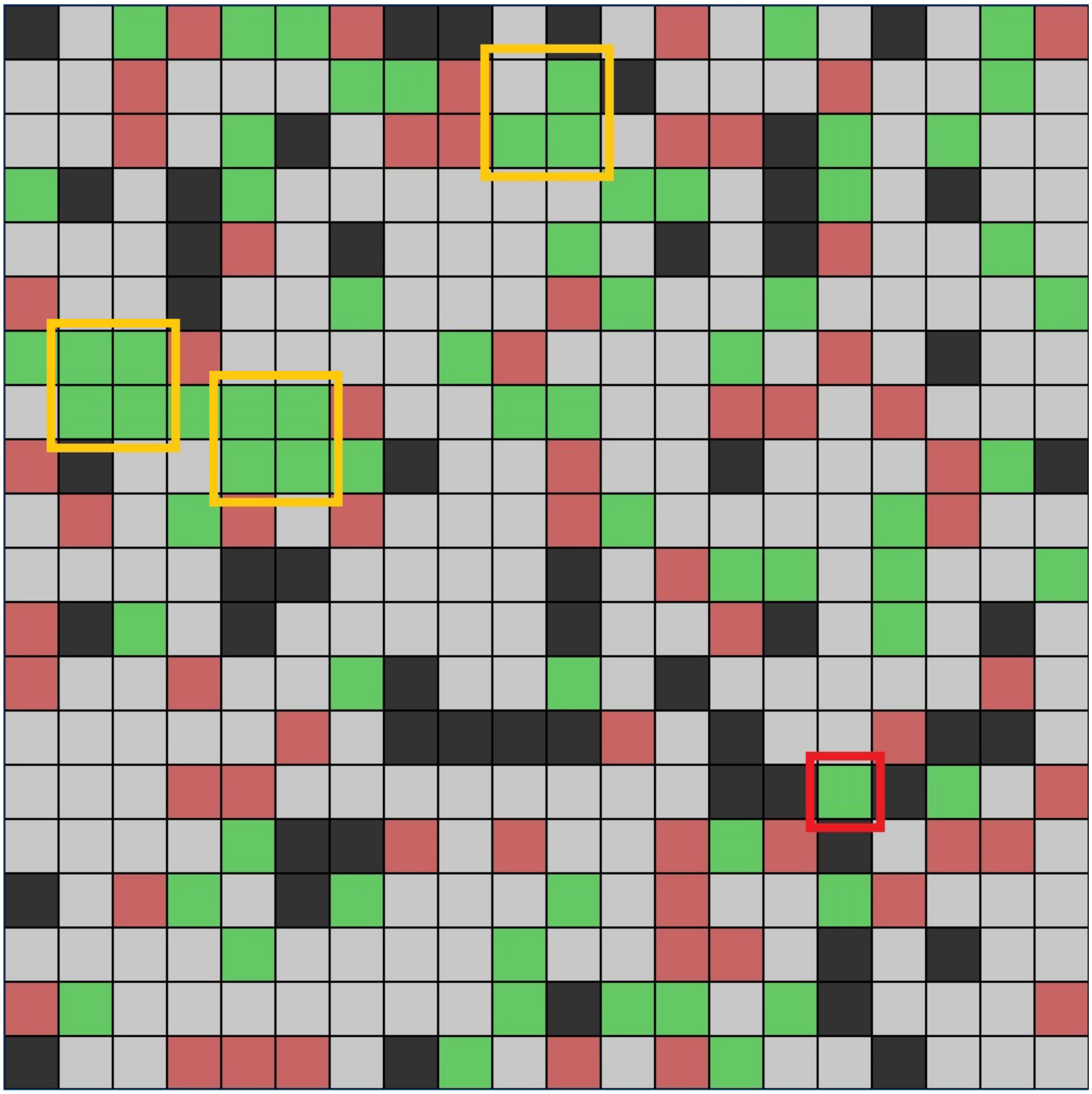
For this task, a 20x20 environment is randomly generated based on some user-defined constants including grid height, grid width, and probabilities of each of the 4 possible states (explained in section 2.1). The dimensions of transition probabilities and rewards is also 20x20. Note that the seed for random has been set to 1 for reproducibility of results. The resulting grid environment is below:



Random grid environment generated with seed=1

5.1 Observations

There is one state in the maze where the agent can receive infinite rewards (marked in red). There are some states where agent can earn reward with high probability (marked in yellow). These states have been highlighted below:



Random grid environment with expected high utility states highlighted

It is expected that these states would have high utilities with optimal action that allows agent to earn maximum reward repeatedly.

5.2 Results of Value Iteration

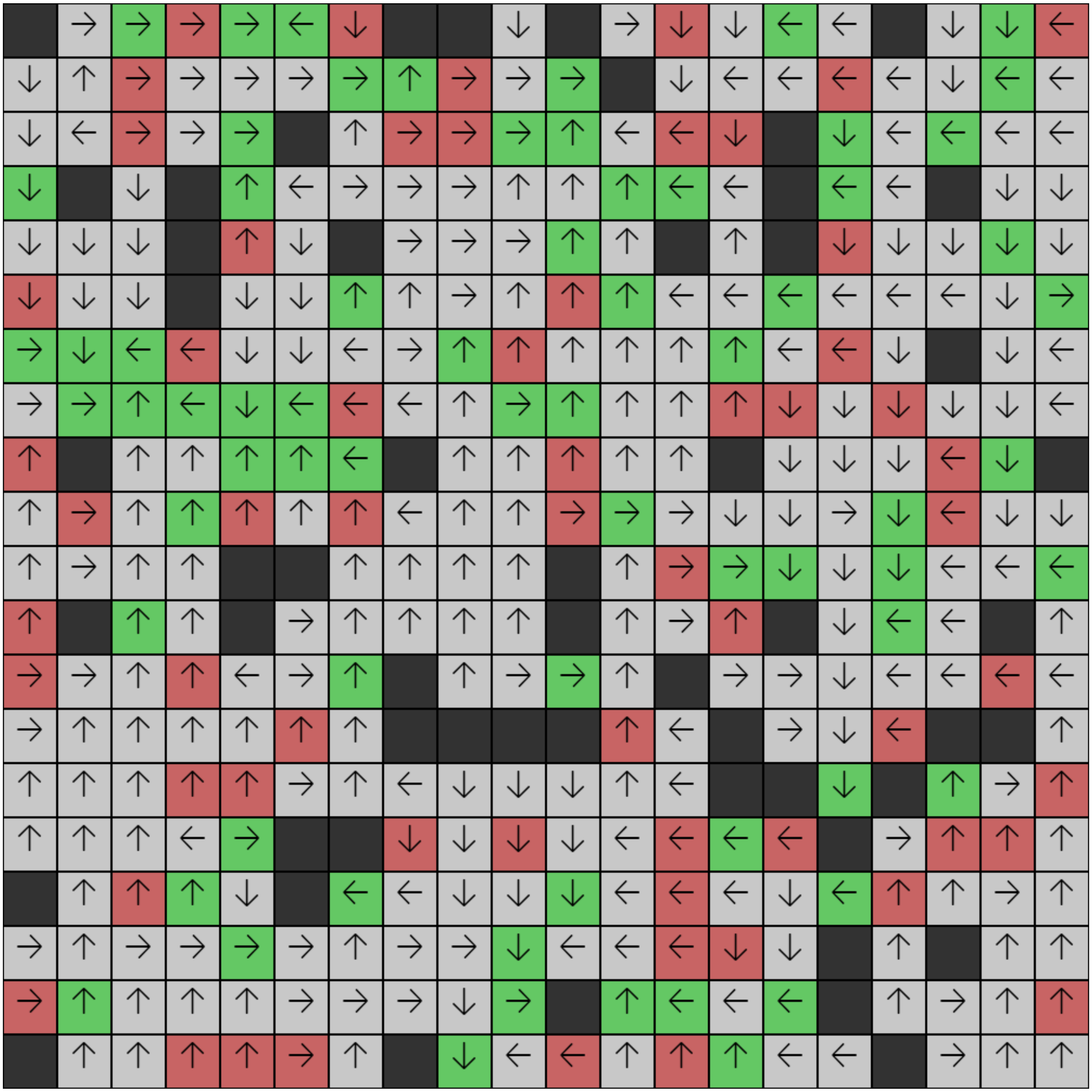
The final utilities calculated by value iteration algorithm with default hyper-parameters is as follows:

	89.447	90.758	90.875	93.272	93.331	92.065			95.654		89.328	90.507	90.604	90.708	89.413		86.251	87.156	84.961
89.014	88.414	88.892	90.990	92.139	93.131	94.333	94.701	94.542	96.914	98.235		93.049	91.799	90.615	88.494	87.455	87.279	87.382	86.092
90.265	89.014	89.332	91.515	92.788		93.102	93.068	95.232	97.807	98.067	96.793	94.432	92.678		89.663	88.392	88.397	87.174	86.009
91.613		91.440		92.719	91.443	92.492	93.635	95.050	96.411	96.755	96.799	96.573	94.892		89.692	88.503		86.395	85.713
91.508	92.508	92.647		90.377	90.881		92.758	93.997	95.252	96.529	95.649		93.658		88.691	88.456	87.428	87.623	86.762
92.590	93.835	93.886		92.238	92.143	92.330	91.753	92.815	93.912	94.190	95.385	94.007	92.874	92.805	91.104	89.724	88.431	87.600	87.802
94.868	95.221	95.129	92.931	93.466	93.323	92.163	91.452	92.645	91.750	92.937	93.901	92.898	92.815	91.543	89.519	89.433		88.629	87.619
93.811	95.217	95.157	94.845	94.782	94.680	92.439	91.259	91.551	92.746	92.955	92.606	91.659	90.469	89.765	90.895	90.603	89.697	89.926	88.629
91.393		93.901	93.750	94.722	94.693	94.237		90.467	91.349	90.696	91.246	90.550		91.933	92.273	93.088	90.740	91.303	
90.192	90.109	92.513	93.566	92.510	93.204	91.820	90.483	89.475	89.987	89.105	91.295	91.324	92.618	93.103	93.430	94.710	92.272	91.265	91.008
89.135	90.092	91.279	92.231			90.483	89.336	88.456	88.775		90.246	91.384	93.941	94.349	94.198	95.110	93.733	92.376	92.177
86.775		91.351	91.065		87.860	89.050	88.184	87.419	87.595		89.208	90.330	91.390		95.306	95.335	94.059		90.977
86.235	88.536	89.853	88.669	87.379	87.633	89.015		86.370	87.119	88.211	88.063		93.903	95.455	96.557	95.336	94.092	91.671	90.402
86.112	87.359	88.432	87.483	86.301	85.622	87.608					85.565	84.334		96.691	98.118	95.613			89.224
85.118	86.184	86.967	85.233	84.214	85.118	86.182	84.951	84.473	84.579	85.249	84.425	83.417			99.901		86.655	85.317	86.693
84.118	85.022	85.628	84.588	84.875			84.099	85.519	85.628	86.564	85.458	83.284	83.460	81.363		82.753	84.073	83.340	85.317
	83.865	83.492	84.576	84.017		87.595	86.247	86.814	87.869	87.964	86.684	84.282	83.210	82.913	83.125	80.962	82.767	82.894	84.059
81.424	82.636	82.779	83.898	85.059	85.148	86.351	86.737	87.901	89.151	87.985	86.834	84.691	83.200	83.948		79.902		81.924	82.845
80.364	82.510	81.888	82.819	83.894	84.537	85.843	87.086	88.281	89.317		86.956	86.620	85.235	85.153		78.928	79.520	80.692	80.578
	81.367	80.856	80.674	81.590	82.428	84.469		89.467	88.412	86.062	85.716	84.524	85.210	84.213	83.112		78.593	79.521	79.522

Plot of Final Utilities calculated by Value Iteration

As per our previous expectations, the final utilities of those states are indeed high. The single state where agent can earn infinite reward has expected utility of 99.9.

The optimal policy for each state based on above utilities is as follows:



Plot of Optimal Policies calculated by Value Iteration

5.3 Results of Policy Iteration

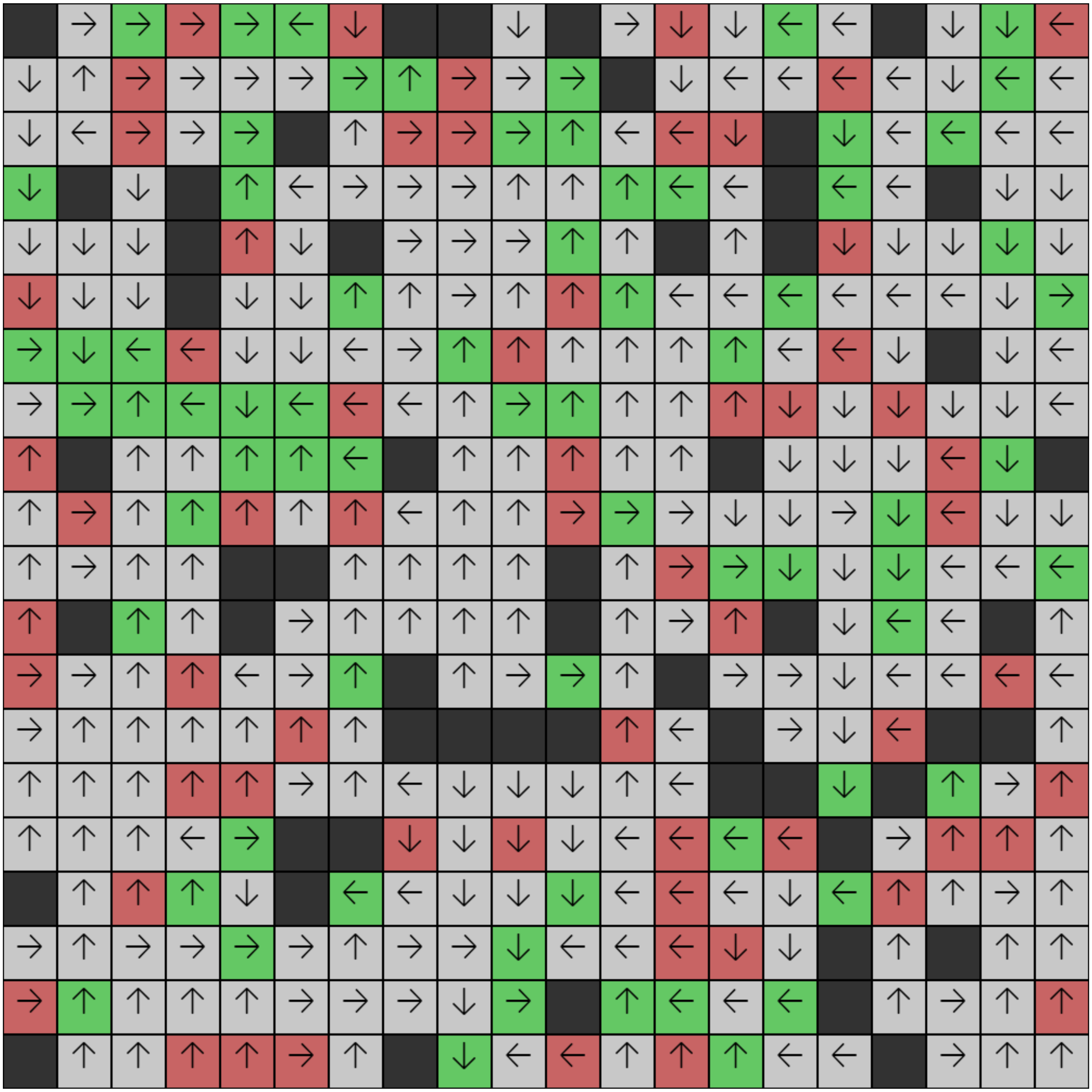
The final utilities calculated by value iteration algorithm with default hyper-parameters is as follows:

	89.545	90.855	90.973	93.370	93.428	92.163			95.752		89.425	90.605	90.702	90.806	89.510		86.349	87.253	85.058
89.109	88.511	88.989	91.087	92.236	93.228	94.431	94.798	94.640	97.011	98.333		93.147	91.896	90.712	88.592	87.553	87.376	87.480	86.190
90.360	89.109	89.429	91.613	92.886		93.200	93.165	95.329	97.904	98.164	96.891	94.530	92.775		89.761	88.490	88.495	87.272	86.107
91.707		91.535		92.817	91.540	92.590	93.733	95.148	96.508	96.853	96.897	96.671	94.990		89.790	88.601		86.494	85.813
91.603	92.602	92.741		90.475	90.976		92.856	94.094	95.349	96.626	95.747		93.756		88.788	88.554	87.526	87.722	86.862
92.685	93.929	93.980		92.332	92.238	92.426	91.850	92.912	94.009	94.288	95.482	94.105	92.971	92.903	91.201	89.822	88.528	87.699	87.901
94.962	95.315	95.224	93.026	93.560	93.418	92.258	91.550	92.743	91.847	93.035	93.999	92.995	92.912	91.640	89.617	89.532		88.728	87.719
93.905	95.312	95.252	94.940	94.877	94.775	92.534	91.354	91.648	92.844	93.053	92.703	91.756	90.567	89.864	90.995	90.703	89.796	90.025	88.728
91.488		93.996	93.845	94.817	94.787	94.332		90.564	91.447	90.794	91.344	90.648		92.032	92.372	93.187	90.839	91.403	
90.287	90.204	92.607	93.661	92.604	93.299	91.915	90.577	89.572	90.085	89.204	91.394	91.423	92.717	93.202	93.529	94.810	92.372	91.365	91.108
89.229	90.187	91.374	92.326			90.577	89.431	88.553	88.872		90.345	91.483	94.040	94.449	94.297	95.209	93.832	92.475	92.276
86.870		91.446	91.160		87.955	89.144	88.279	87.515	87.692		89.307	90.429	91.490		95.405	95.435	94.158		91.076
86.330	88.631	89.947	88.763	87.474	87.728	89.109		86.467	87.217	88.310	88.162		94.002	95.554	96.656	95.435	94.191	91.770	90.501
86.206	87.454	88.526	87.578	86.395	85.716	87.703					85.664	84.433		96.791	98.218	95.712			89.323
85.213	86.279	87.062	85.327	84.308	85.213	86.276	85.045	84.563	84.668	85.339	84.523	83.515			100.000		86.754	85.416	86.793
84.212	85.116	85.723	84.682	84.967			84.188	85.608	85.717	86.653	85.548	83.374	83.551	81.453		82.852	84.172	83.439	85.416
	83.959	83.587	84.670	84.106		87.684	86.336	86.903	87.958	88.053	86.773	84.371	83.299	83.002	83.214	81.060	82.866	82.994	84.158
81.518	82.730	82.869	83.988	85.148	85.236	86.440	86.825	87.990	89.239	88.074	86.923	84.780	83.289	84.037		80.000		82.023	82.945
80.457	82.603	81.978	82.908	83.983	84.625	85.931	87.175	88.370	89.405		87.044	86.709	85.324	85.242		79.026	79.620	80.792	80.678
	81.460	80.947	80.763	81.679	82.517	84.558		89.556	88.501	86.151	85.805	84.613	85.299	84.302	83.201		78.692	79.620	79.622

Plot of Final Utilities calculated by Policy Iteration

As per our previous expectations, the final utilities of those states are indeed high. The single state where agent can earn infinite reward has expected utility of 100.0.

The optimal policy for each state based on above utilities is as follows:



Plot of Optimal Policies calculated by Policy Iteration

5.4 Impact of the Number of States on Convergence Rate

Computational complexity of value iteration algorithm is quadratic with the number of states, and policy iteration algorithm terminates in at most exponential number of iterations. Thus, increasing the number of states will exponentially increase the running time for both algorithms (on average).

A possible optimization of policy iteration algorithm involves pick a subset of states and applying policy improvement to that subset. This is called asynchronous policy iteration and guarantees convergence to optimal policy. One additional advantage of this is the flexibility to design efficient heuristic algorithms that concentrate on updating values of states that are likely to be reached by a good policy.

5.5 Impact of Discount Factor on Convergence Rate

Another important parameter affecting convergence rate is the discount factor. Discount factor is associated with time horizons and determines how much the agent cares about rewards in the distant future compared to immediate future. This introduces some kind of a trade-off between the optimal solution and the fastest solution.

In the current task, we are solving an infinite horizon MDP where utility of state is the sum of discounted rewards obtained if policy π is followed. Thus, expected utility of state is:

$$U^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t)\right]$$

By selecting a discount factor $\gamma < 1$, the sum converges, and the algorithm is guaranteed to find the optimal policies. If instead $\gamma = 1$, the sum would not converge and hence would not be a good optimization criterion.

5.6 Impact of Maze Complexity on Learning the Right Policy

Both value iteration and policy iteration are guaranteed to optimal policy for discounted MDP with finite states and infinite time horizon. This implies that the greedy policy learned by the agent will also be optimal. However, if the environment was made in such a way that it has infinite state space or infinite action space, then the algorithms would not guarantee to converge.