CZ4046: Intelligent Agents
Assignment 1

Wong Yuh Sheng Reuben
U1820016H

16 March 2021

Contents

1 Introduction

In this assignment, we were required to implement 2 learning algorithms for solving Markov Decision Processes (MDPs), namely **(1) Value Iteration** and **(2) Policy Iteration**. In this report, we will first go through the implementation of the maze learning environment as well as any details that need to be made known regarding them. Next, we will move on to describing the implementations of the aforementioned MDP solvers.

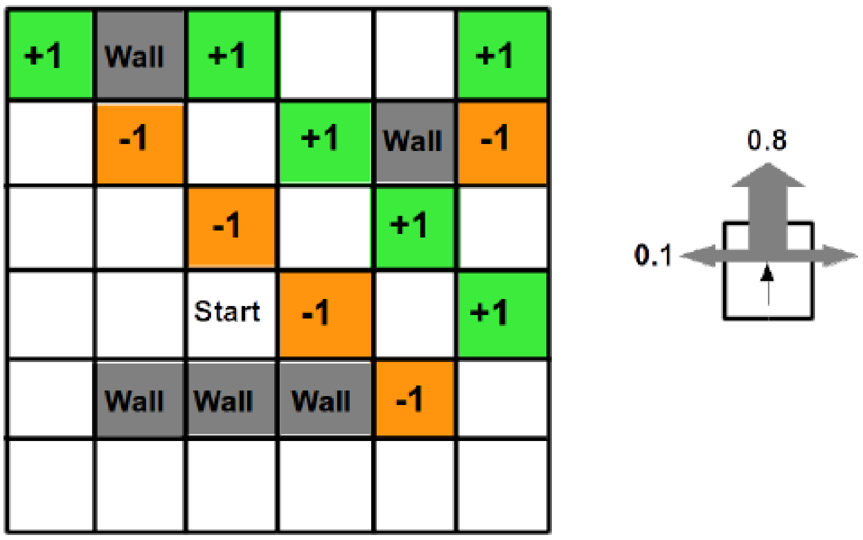


Figure 1: Visual Depiction of Problem Setting

Figure 1 provides a visual depiction of the environment of interest. In this environment, we have a total of 4 different tile types. Green tiles represent reward tiles with +1 reward, while orange tiles represent punishment tiles with -1 reward. White tiles simply have a -0.04 reward while wall tiles are impassable objects. As seen on the right side of Figure 1, intended actions are only realised with a probability of 0.8, and we have a combined probability of 0.2 to take actions orthogonal to the intended action.
To run the code, use `python main.py`. `vtheta`, `ptheta` and `difficulty` are optional flags.

2 Environment

2.1 Maze Template

The environment is implemented as a 2-dimensional matrix in numpy, containing tile objects that encapsulate the various attributes of the tiles.

```
2, 1, 2, 0, 0, 2
0, 3, 0, 2, 1, 3
0, 0, 3, 0, 2, 0
0, 0, 0, 3, 0, 2
0, 1, 1, 1, 3, 0
0, 0, 0, 0, 0, 0
```

Figure 2: Example Maze Template

After providing a maze template, as in Figure 2, with natural numbers representing the type of tile to be created at that location, the maze program will then scan through the template and create the appropriate tiles in the numpy matrix.

2.2 Maze and Tiles

To create the maze, the maze program relies on the `TileFactory` class to instantiate the appropriate `Tile` objects.

```python
class TileFactory(object):
    def create_tile(self, map_value: int, location: tuple) -> Tile:
        if map_value == 0:
            return WhiteTile(map_value, location)
        elif map_value == 1:
            return GreyTile(map_value, location)
        elif map_value == 2:
            return GreenTile(map_value, location)
        elif map_value == 3:
            return OrangeTile(map_value, location)
        else:
            raise ValueError("Values in map restricted to 0, 1, 2 or 3 only")

class Tile(object):
    def __init__(self, map_value:int, location: tuple):
        self.map_value = map_value
        self.location = location
        self._set_attributes()

    def _set_attributes(self):
        raise NotImplementedError

class WhiteTile(Tile):
    def __init__(self, map_value: int, location: tuple):
        super(WhiteTile, self).__init__(map_value, location)

    def _set_attributes(self):
        self.color = COLORS["white"]
        self.passable = True
        self.reward = -0.04
        self.learnable = True
```

Listing 1: TileFactory and Tile Objects

```python
Class Maze:
    def transitions(self, state, action):
        location = self._to_location(state)
        transitions = []
        majority_location = self.get_new_location(action, location)
        majority_probability = 0.8
        majority_transition = [self._to_state(majority_location),
    majority_probability]
        transitions.append(majority_transition)

        # transitions if movement is orthogonal to selected action
        minority_probability = 0.1
        orthogonal_actions = self.action_orthogonals[action]
        ortho_action_1 = orthogonal_actions[0]
        minority_location_1 = self.get_new_location(ortho_action_1, location)
        transitions.append([self._to_state(minority_location_1),
    minority_probability])
        ortho_action_2 = orthogonal_actions[1]
        minority_location_2 = self.get_new_location(ortho_action_2, location)
        transitions.append([self._to_state(minority_location_2),
    minority_probability])

        return np.array(transitions, dtype=object)
```

Listing 2: Maze

Listing 1 shows snippets of code that describe the core tile creation process, as well as the attributes that are held within a tile. Meanwhile, Listing 2 shows the core piece of code within the `Maze` class. Crucially, this `transitions` method provides the transition probabilities to the learner given a `(state, action)` tuple, with which the learner would use to update its utilities in expectation. Please see `env/maze.py` and `env/tiles.py` in the provided source code for more details.

3 Learners

Two learners have been implemented and will be described in this report. Briefly, **Value Iteration** aims to learn an *optimal value function* with which it will then use to obtain an *optimal policy*. This process is based on the *bellman optimality operator*.
On the other hand, **Policy Iteration** is based on the *bellman operator*, and first evaluates a given policy, iterating through this process until the utilities have converged sufficiently. It then proceeds to improve its policy given these utilities. However, at this juncture, the previous utility values are no longer accurate, since the input policy has now been changed. Policy iteration then has to cycle through these processes multiple times until the policy no longer changes, and we can then conclude the policy iteration process.

3.1 Value Iteration

3.1.1 Algorithm

As described above, we will iteratively apply the bellman equation (Eq. 1) to obtain utility estimates for each *learnable* state (non-wall states) in our environment.

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

Algorithm 1: Value Iteration

```
Function BellmanOptimalityUpdate(state, gamma, V_old):
    Q_values = empty list
    reward = env.get_reward(state)
    for action ∈ $\mathcal{A}$(state) do
        q_value = 0
        for $s'$, probability ∈ environment transitions do
            q_value += probability * (gamma * V_old[$s'$])
        end
        Q_values.append(q_value)
    end
    self.V[s] = reward + max(Q_values)
Function ValueIteration(gamma, theta):
    while True do
        $\Delta \leftarrow 0$          /* to compare against theta as stopping criterion */
        V_old ← self.V          /* cache utility table for performing updates */
        for $s \in \mathcal{S}$ do
            v = V_old[s]          /* cache value for comparison later */
            BellmanOptimalityUpdate(s, gamma, V_old)
            $\Delta$ ← max($\Delta$, abs(v - self.V[s]))
        end
        if $\Delta$ < theta then
            break          /* exit the loop within theta margin */
        end
    end
    GreedifyPolicy()    /* recover policy, made greedy with respect to utilities */
```

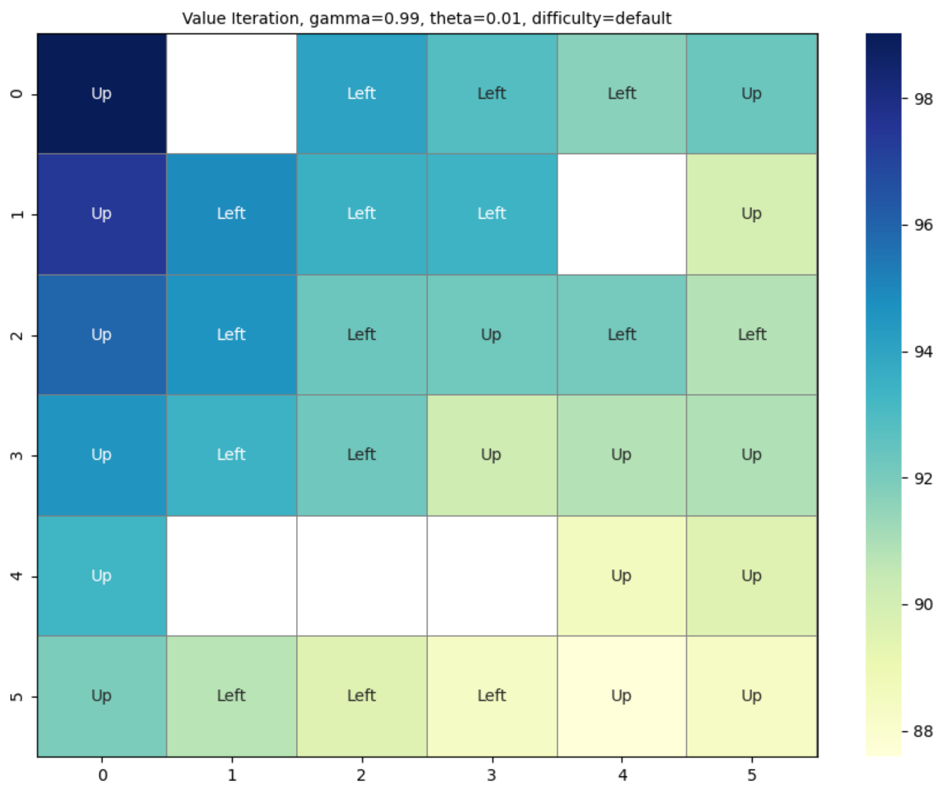


Figure 3: Optimal Policy for Value Iteration

Please see `learner/learner.py` and `learner/value_iteration.py` for more details regarding the learner base class and implementation details of value iteration respectively.
Algorithm 1 details the iterative process for finding the optimal utility values, and then recovering the optimal policy from them. There are 2 key components of this algorithm.
First, we define a threshold *theta*, representing our criterion, that when the maximal change in utility values is less than theta, we stop our value iteration.
Second, the `BellmanOptimalityUpdate` which is the core of the learning process, computing utility values given the environment transitions. It is important to note that the utilities have to be cached when performing the update, such that we do not bootstrap of utility values that we **have just updated** within the same iteration. As the update is a dynamic programming approach, we perform the update for all states stored in our table.
We can break down this update into 2 steps, **(1)** we consider all possible actions that we can take from the state, **(2)** and then for each one of these actions, we compute the expectation of utilities over all possible successor states given `(s,a)`.

3.1.2 Recovered Optimal Policy

Figure 3 details the optimal policy that was recovered after performing value iteration to acquire the optimal utility values. The positions of the tiles correspond exactly to our constructed maze, whilst the colors represent the utility values (with darker blue signifying higher values). Intuitively, we can see the policy corresponds nicely to what we would do as humans. The policy directs us to the 3 green tiles on the top most row, where we can deliberately take actions that would allow us to maximise our chances of remaining in those tiles, and hence accruing more reward. Moreover, we can clearly see that state `(0, 0)` is the best state to be in, as selecting the action `Up` would ensure that we would continue to stay in that state (since there is a boundary on the left and a wall on the right), permanently acquiring rewards of 1.

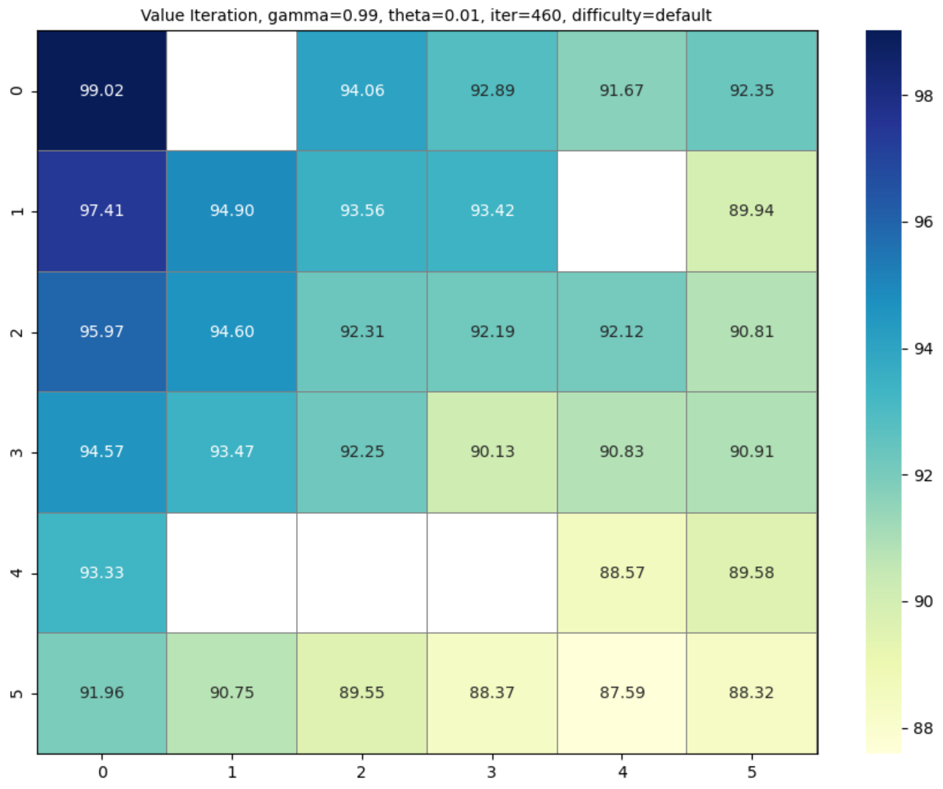


Figure 4: Final Utility Values for Value Iteration

3.1.3 Converged Utility Values

Figure 4 depicts the utilities that we have obtained at convergence, i.e. when the maximal change in utility values between iterations are lower than the value of 0.01 that we had set for theta. A darker blue denotes a higher utility value, with the top leftmost tile obtaining the highest utility of 99.02. These values were obtained with a discount factor of 0.99, which is a fairly long time horizon.

3.1.4 Utility Values as Function of Iterations

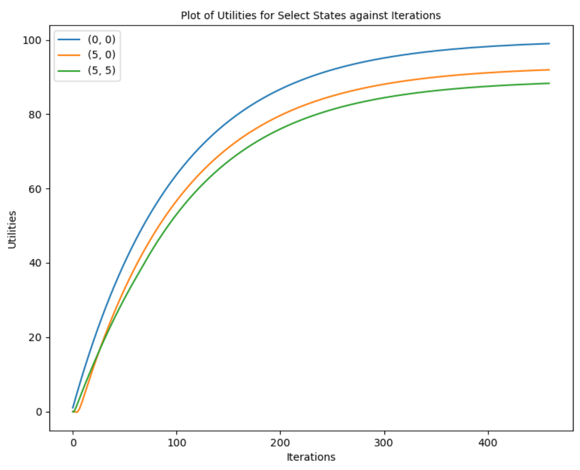
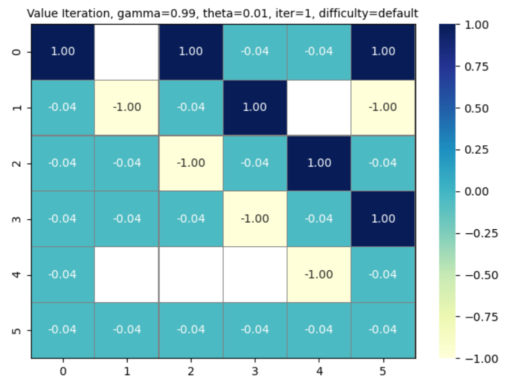
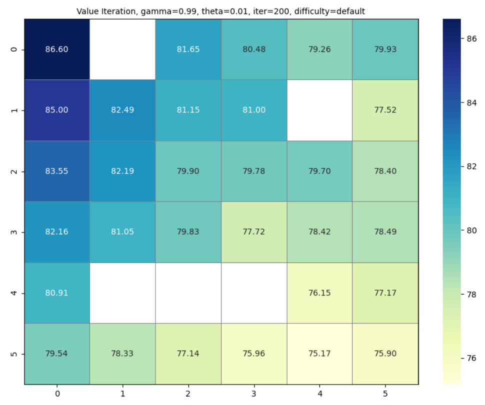


Figure 5: Utilities as a Function of Iterations

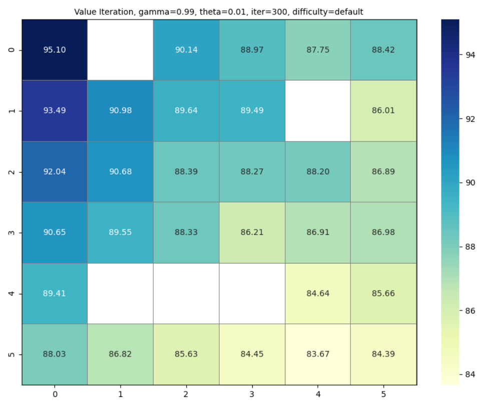
Of course, as we increase the number of iterations, the utility values will converge to their true value. As mentioned before, in this implementation, we use theta as a hyperparameter that may be tuned. This value theta determines our stopping condition, as explained in relation to Algorithm 1. We can then plot the utilities as a function of the number of iterations, and observe how their values converge towards the true values. As we have many states that become unwieldy to keep track of, we will only plot the utilities as specified in the assignment for 3 states, `(0, 0)`, `(5, 0)` and `(5, 5)`.



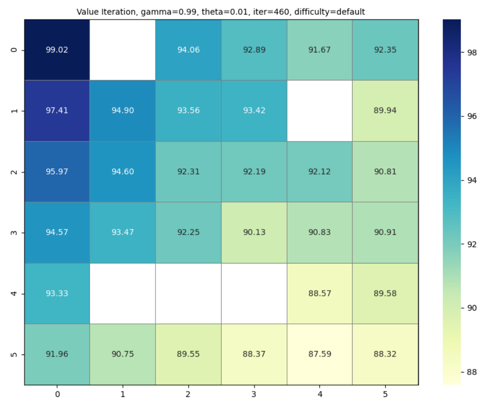
(a) Utilities at Iteration 1



(b) Utilities at Iteration 200



(c) Utilities at Iteration 300



(d) Utilities at Final Iteration (460)

Figure 6: Evolution of Utilities Presented in Grid Format

This report also provides an intuitive visualisation of the evolution of utility values with iterations, albeit at a much coarser granularity. Figure 6 shows the utilities for all states, visualised as a gridworld, for select iterations, namely, at iterations 1, 200, 300 and 460. It is interesting to note that only the first iteration is extremely distinct from the rest, having only obtained information from their initial immediate rewards. By the 200th iteration, the relative values and the trends between these values have stabilised (as observed from the relative intensities between tiles on the heatmap), and only continue to iterate for the next 260 iterations to arrive closer at their true values.

3.2 Policy Iteration

Please see `learner/policy_iteration.py` for more implementation details.

3.2.1 Algorithm

Similar to Value Iteration in Equation 1, policy iteration also updates its utility estimates in an iterative fashion. However, policy iteration more closely resembles for framework of **Generalised Policy Iteration**, which alternates between policy evaluation and policy iteration until a convergence of the policy. The updates made at each iteration also defers slightly, and is presented in Equation 2.

$$U_{i+1}(s) \leftarrow R(s) + \gamma \sum_{s'} P(s'|s, \pi_i(s))U_i(s') \tag{2}$$

The key difference here, is that while Equation 1 computes the summation over all possible actions $a \in \mathcal{A}(s)$ and takes the maximal of such sums, Equation 2 only computes the sum for **a single action** specified by the policy at that state, $\pi_i(s)$. It is for this reason that policy iteration tends to converge more quickly.

Algorithm 2: Policy Iteration

```
Function ModifiedBellmanUpdate(s, gamma, V):
    Input: Policy at Iteration i, $\pi_i$
    reward = env.get_reward(s)
    action = $\pi_i(s)$
    inner_sum = 0
    for $s'$, probability ∈ environment transitions do
        /* V is a cache of self.V */
        inner_sum += probability * (gamma * V[$s'$])
    end
    self.V[s] = reward + q_value
Function PolicyEvaluation(gamma, theta):
    Input: Policy to be evaluated at Iteration i, $\pi_i$
    while True do
        $\Delta \leftarrow 0$
        V_old ← self.V
        for $s \in \mathcal{S}$ do
            v = V_old[s]
            ModifiedBellmanUpdate(s, gamma, V_old)
            $\Delta \leftarrow$ max($\Delta$, abs(v-self.V[s]))
        end
        if $\Delta <$ theta then
            break
        end
    end
Function PolicyImprovement():
    Set: policy_stable = True
    for $s \in \mathcal{S}$ do
        // if we would not have taken the maximal action under $\pi$, then update $\pi$ to do so
        if $\max_{a \in \mathcal{A}(s)} \sum_{s'} P(s'|s,a)U(s') > \sum_{s'} P(s'|s,\pi(s))U(s')$ then
            $\pi(s) \leftarrow \text{argmax}_{a \in \mathcal{A}(s)} \sum_{s'} P(s'|s,a)U(s')$
            policy_stable = False
        end
    end
Function PolicyIteration(gamma, theta):
    policy_stable = False
    while not policy_stable do
        // PolicyIteration cycles between evaluation and improvement
        PolicyEvaluation(gamma, theta)  // note this is itself an iterative process
        policy_stable = PolicyImprovement()
    end
```

Algorithm 2 describes the implementation of `PolicyIteration`, which cycles between policy evaluation and policy improvement until there is no more change in the policy, and we have obtained the optimal policy. The fundamental difference between Policy Iteration and Value Iteration is captured in the `ModifiedBellmanUpdate` function. Unlike Value Iteration, which considers all possible actions $a \in \mathcal{A}(s)$, and takes the max computed, we see clearly here that the action is given to us by our policy $\pi_i(s)$. Thus, we only compute the utility for the state as determined by the policy.

Similar to Algorithm 1, the `ModifiedBellmanUpdate` seeks to learn the utility values for all states, with the key difference being that it learns utility values for a given policy, instead of the optimal utility values.

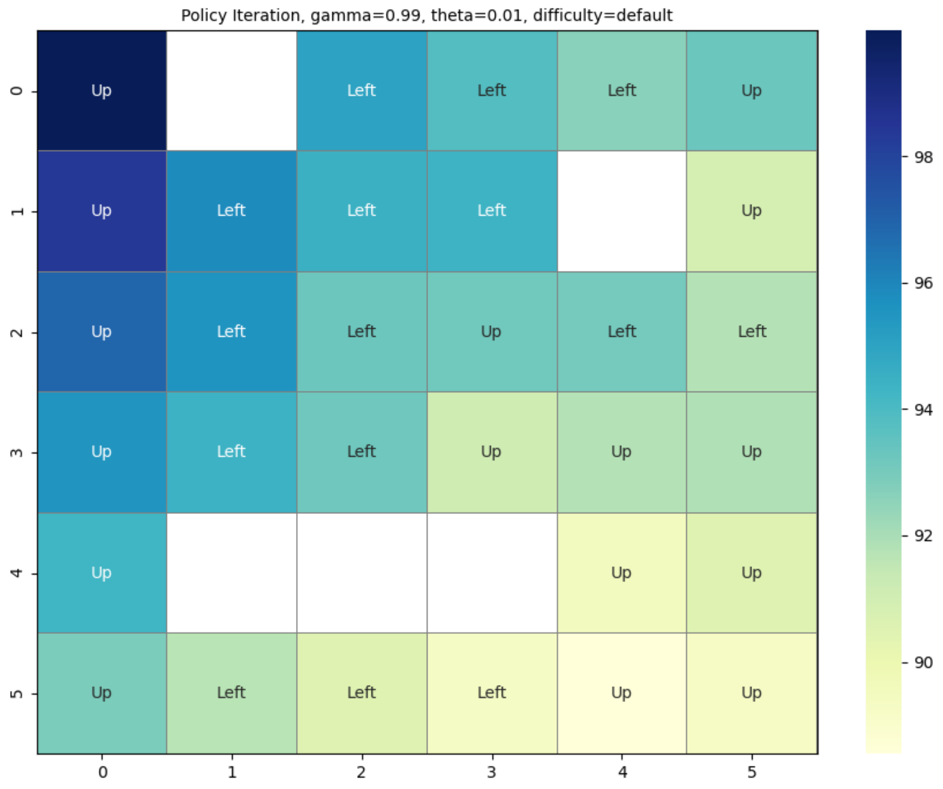


Figure 7: Optimal Policy for Policy Iteration

Also of note, the condition for improving the policy is when the expected utility of the maximal action is greater than the expected utility of the action we took given our policy. In such a scenario, we then improve the policy greedily with respect to our utility values. Because our policy has been changed, we update the flag `policy_stable` to be false.

3.2.2 Recovered Optimal Policy

Figure 7 provides a visual depiction of the optimal policy derived at the end of the policy iteration process, when there were no more changes to the policy and the policy was deemed to have converged.

To avoid cluttering the main report, please see Figure A.3 of Appendix A for a plot of the policies at different stages of improvement.

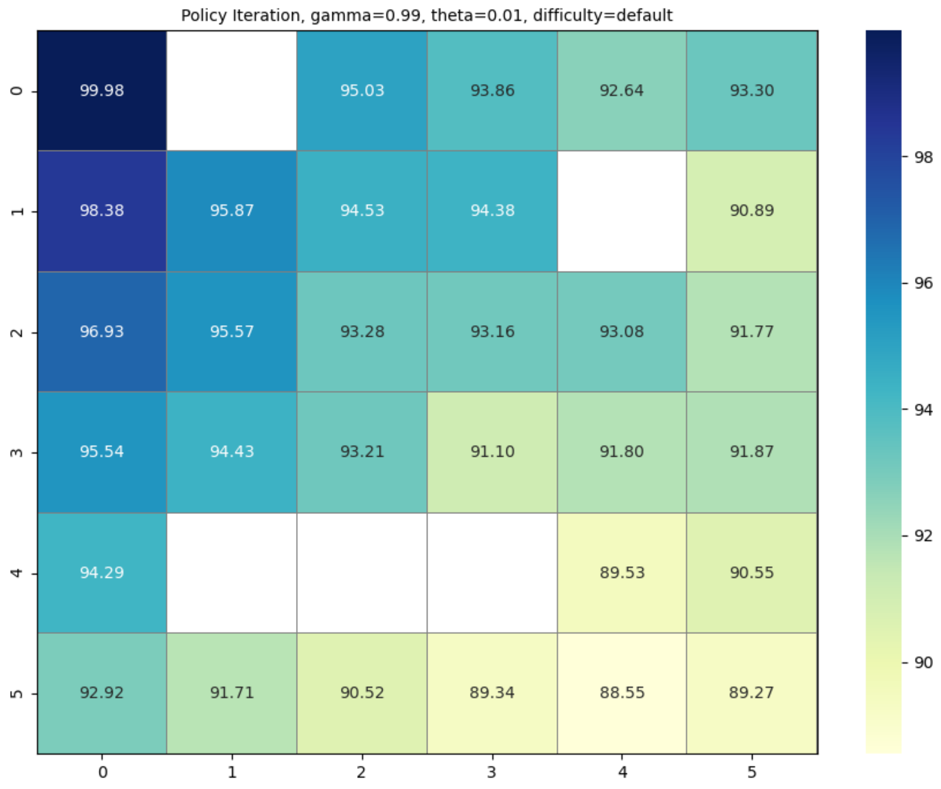


Figure 8: Final Utility Values for Policy Iteration

3.2.3 Converged Utility Values

Figure 8 provides a visual depiction of the utility values obtained at convergence for a theta of 0.01 and discount factor of 0.99. I would like to point out that the converged utility values for policy iteration and value iteration should **be the same**. However, this discrepancy is due to setting our stopping condition of theta to be 0.01, being small enough such that they are close enough to their true values, but yet not exactly their true values. This is in the interest of obtaining an estimate that is good enough, and saves time. However, for completeness, please see Figure A.1 of Appendix A for plots of utilities with a much smaller value of theta, and the much more similar utility values for both methods. At this juncture, I would also like to point out that setting too small a value of theta would do little to improve the final result, and that differences in values for the less significant values can be due to numerical instability.

3.2.4 Utility Values as Function of Iterations

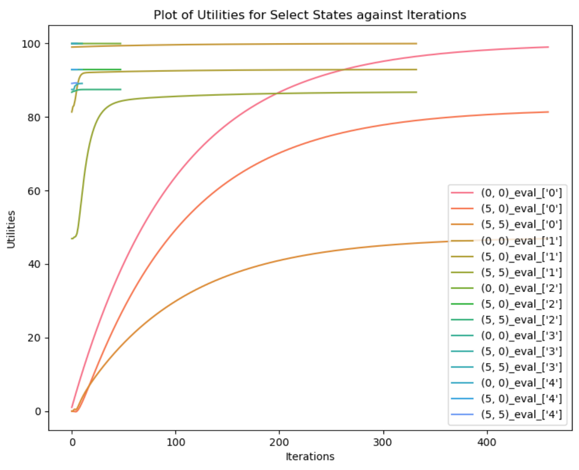


Figure 9: Utilities as a Function of Iterations

Like before, we keep track of the utilities for the same states that we had kept track of in Value Iteration. On first glance, Figure 9 looks peculiar as the line plots have differing lengths and shapes. Upon closer inspection, we can see that policy evaluation was performed 5 times in the policy iteration process. Thus, the 15 curves presented here are from those 5 evaluations, for our 3 states of interest.

The first evaluation took the longest to converge within our set margin of theta as the utilities were the furthest from their through values since they had been initialised to 0. Hence, it required more than 400 iterations to converge sufficiently. Subsequently, having improved the policy, the initial evaluations were no longer accurate and the utilities had to be determined again. However, the utilities were now closer to their true values, and converged in a shorter amount of time. This accounts for the set of curves terminating near the 300th iteration as their starting values were close to the true values. This also explains their significantly less steep curves. This then repeats itself for the remaining evaluations, giving us the set of curves we see in Figure 9.

While there are too many plots for the utilities to show their evaluation within a single policy evaluation, we may observe the changes in their values between policy evaluations. This is presented in Figure A.2 of Appendix A.

4 Harder Maze

4.1 Maze Template

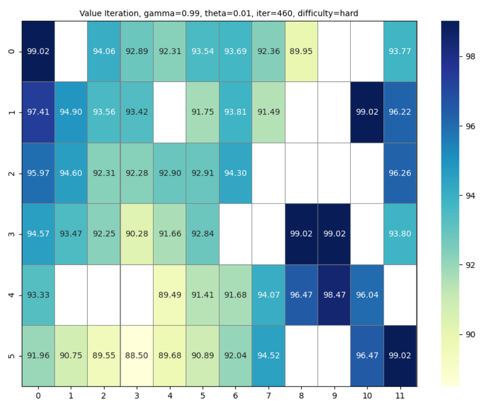
Due to the design of the code base, it is extremely easy to extend its usage to various maze designs, and we can easily build a new maze by supplying a new template.

```
2, 1, 2, 0, 0, 2, 2, 0, 3, 1, 1, 3
0, 3, 0, 2, 1, 3, 2, 3, 1, 1, 2, 3
0, 0, 3, 0, 2, 0, 2, 1, 1, 1, 1, 2
0, 0, 0, 3, 0, 2, 1, 1, 2, 2, 1, 3
0, 1, 1, 1, 3, 0, 3, 3, 3, 2, 3, 1
0, 0, 0, 0, 0, 0, 3, 2, 1, 1, 3, 2
```

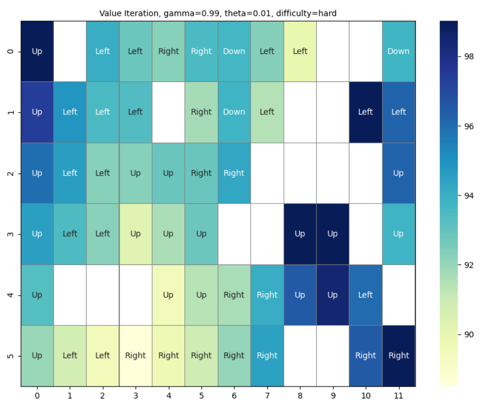
Figure 10: Maze Template for the Harder Maze

Figure 10 shows the gridworld design that was used in this study. See `env/mazes/hard.txt` for this template. Custom designs can be placed into this directory as well.

4.2 Value Iteration Results



(a) Final Utilities for Value Iteration in the Hard Environment

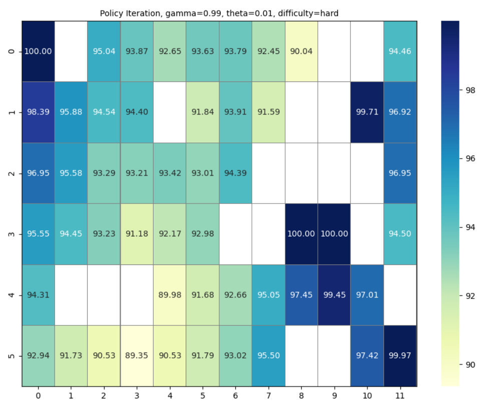


(b) Optimal Policy for Value Iteration in the Hard Environment

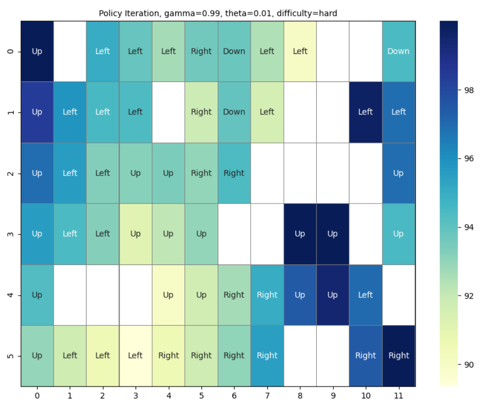
Figure 11: Results for Value Iteration in the Hard Environment

The concepts and explanations mentioned before carry over here as well. We can inspect the plots in Figure 11 to see that the optimal policy concurs with our intuition. In states that are "boxed in" like `(0, 0)` and `(3, 8)`, we see that the `Up` action always allows us to stay in states with +1 reward. Hence, even in this more complicated environment, Value Iteration still finds the optimal policy.

4.3 Policy Iteration Results



(a) Final Utilities for Policy Iteration in the Hard Environment



(b) Optimal Policy for Value Iteration in the Hard Environment

Figure 12: Results for Policy Iteration in the Hard Environment

Figure 12 depicts the converged utilities and optimal policy in this hard environment. Like before, we have obtained the same policy as in Value Iteration and our utility values are extremely as well. Differences can be attributed to our choice for the value of theta and numerical instability.

4.4 Convergence of Value and Policy Iteration

Our discussion of the convergence of Value and Policy Iteration towards optimal utilities and policies despite varying environment complexities is directly tied to their convergence properties. First, we shall view the Bellman update as an operator B that simultaneously updates the values for all states, i.e.

$$U_{i+1} \leftarrow BU_i$$

Next, we measure the "length" of our utility vector as the maximal value of the absolute of its components.

$$||U|| = \max_s |U(s)|$$

The distance between utility vectors $||U - U'||$ is then the maximal difference between any 2 corresponding elements. Then, for any 2 utility vectors U and U', we have

$$||BU_i - BU_i'|| \leq \gamma ||U_i - U_i'||. \tag{3}$$

This shows that the Bellman update is a contraction by a factor of γ on the space of utility vectors. From the properties of a contraction that it has only 1 fixed point, and that **repeated applications of the function will always reach this fixed point in the limit**.

While there are means to determine appropriate error bounds for the algorithms, it is not of interest now. More importantly, the statement above shows that Value Iteration and Policy Iteration will always converge to the true utilities and thus, optimal policies, in spite of whatever environment we throw at it.

Hence, to clearly address the questions brought up in the assignment, the number of states and complexity of the environment may affect the speed of convergence, but Value Iteration and Policy Iteration will **always** be able to converge, as long as the environment is *finite.* Therefore, given the constraint of a finite environment, we can make it as complicated as we want and still be always able to learn the right policy.

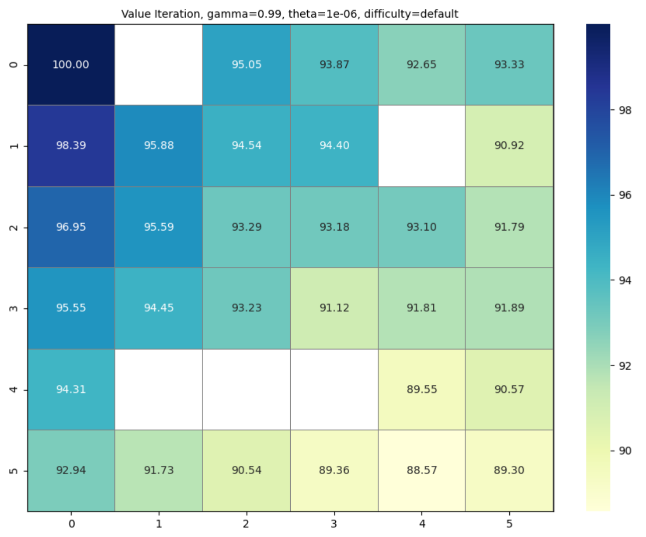
5 Conclusion

This report first introduces the maze environment for learning and briefly discusses its design and properties. The report then explains in great detail, the implementations and results of Value Iteration and Policy Iteration. Important trends in utilities and policy are clearly depicted in the given plots. Any additional graphs can be found within the `visualisations` directory of the given source code. Additional plots are also provided in Appendix A for more completeness.

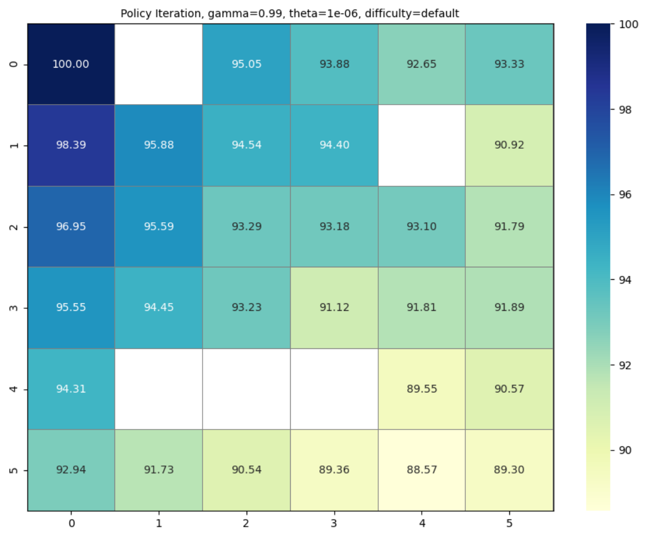
Lastly, we discussed the potential implications of environment design on the abilities of Value Iteration and Policy Iteration to learn the right policy. In this section, we provide a sample maze that was more complex than the one provided and show that the optimal policy was still learnt. Moreover, we then show that these algorithms will converge to a fixed point as the Bellman Update is a contraction. We then conclude that these algorithms will always learn the right policy, no matter the complexity of the environment, so long as it is *finite.*

A Appendix

This appendix contains various plots that may have cluttered the main report, but could provide a more complete picture of the findings.

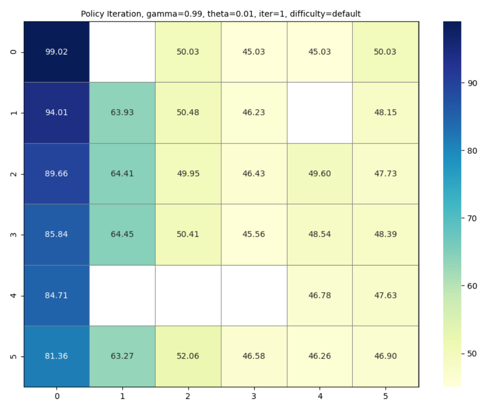


(a) Utilities for Value Iteration with Theta=10^{-6}

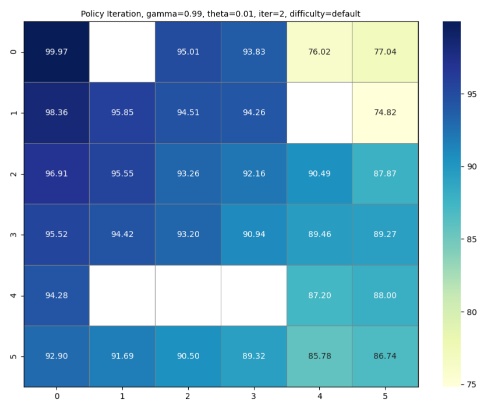


(b) Utilities for Policy Iteration with Theta=10^{-6}

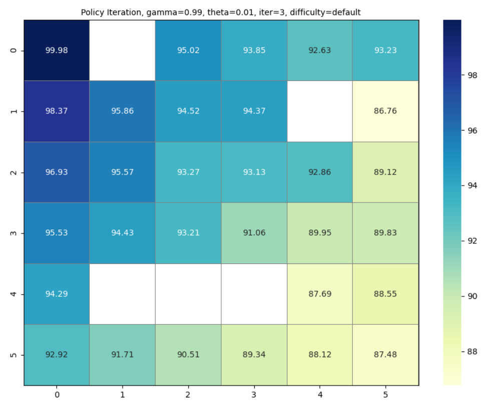
Figure A.1: Plots of Utilities with Smaller Theta



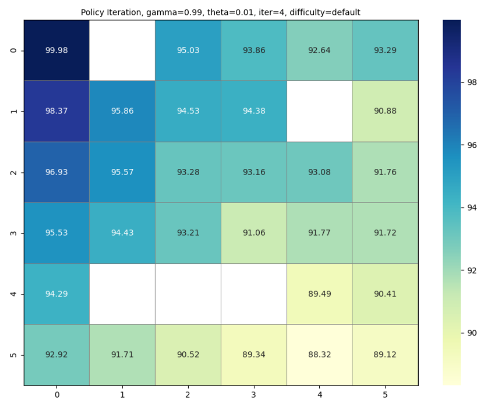
(a) Utilities after Policy Evaluation 1



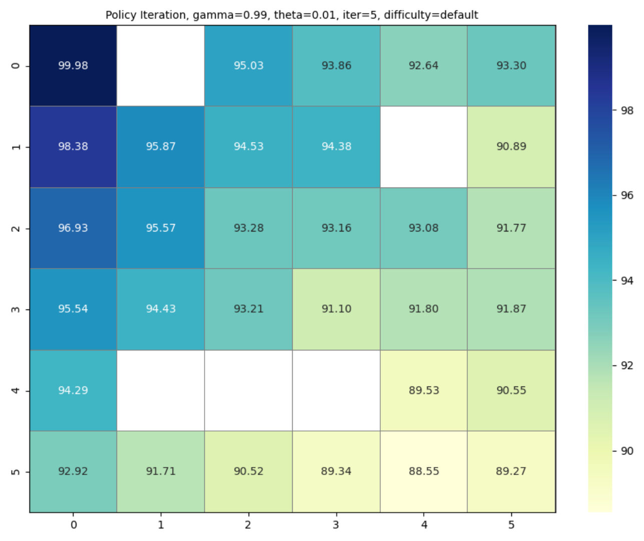
(b) Utilities after Policy Evaluation 2



(c) Utilities after Policy Evaluation 3

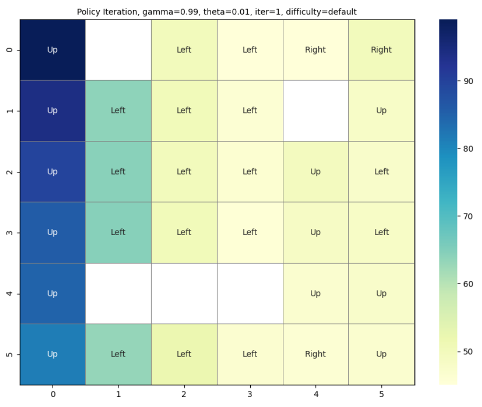


(d) Utilities after Policy Evaluation 4

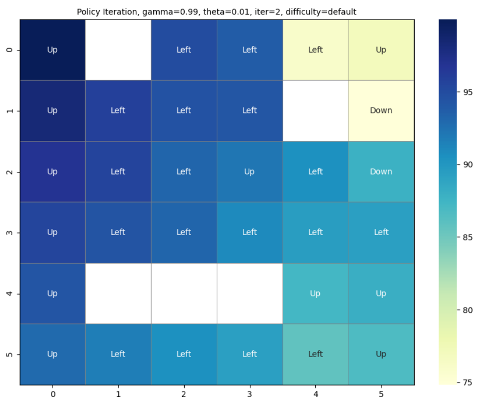


(e) Utilities after Policy Evaluation 5

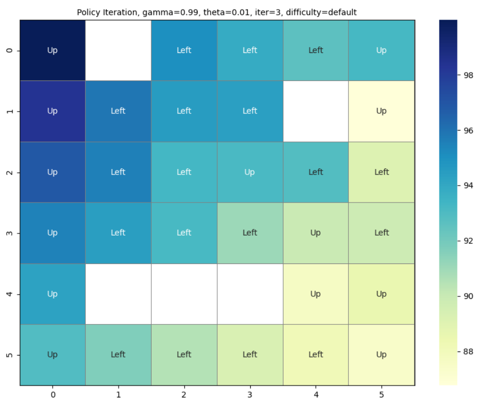
Figure A.2: Evolution of Utilities Presented in Grid Format



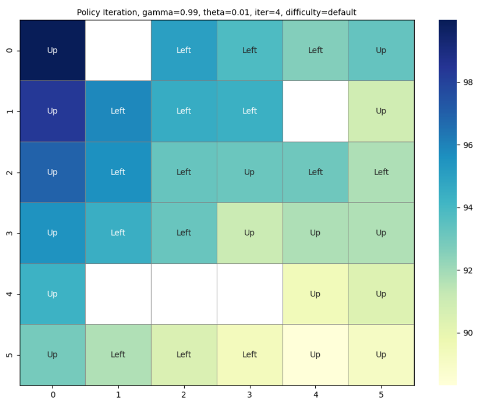
(a) Policy after Policy Improvement 1



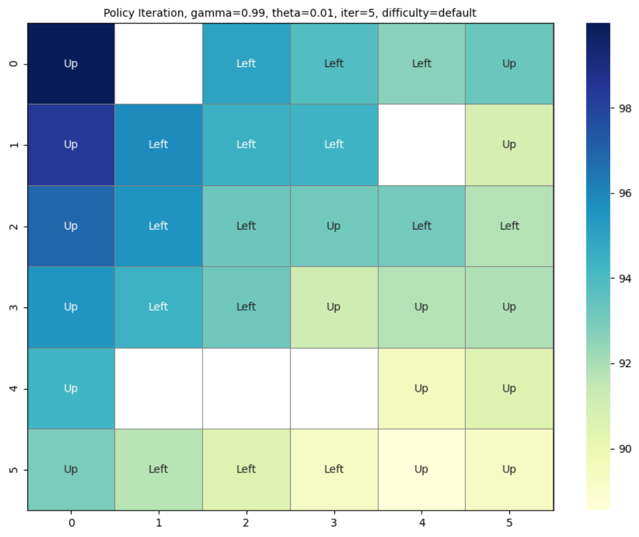
(b) Policy after Policy Improvement 2



(c) Policy after Policy Improvement 3



(d) Policy after Policy Improvement 4



(e) Policy after Policy Improvement 5

Figure A.3: Evolution of Policies Presented in Grid Format