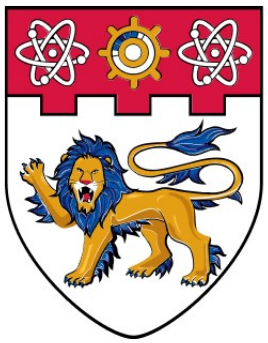

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

Nanyang Technological University
School of Computer Science and Engineering

Intelligent Agents (CZ4046)
Assignment 1 Report

Author - Singh Jasraj
Matriculation Number - U1940558D
Course Index - 10495

Contents

1 Introduction

1.1 Problem Statement

Given a maze environment, model the problem of finding utility of each state using a Markov Decision Process (MDP), and solve it using Value Iteration and Policy Iteration algorithms. Use these estimates to determine an optimal policy for your agent. Furthermore, analyse how the utility values of different states evolve over time.

1.2 Problem Description

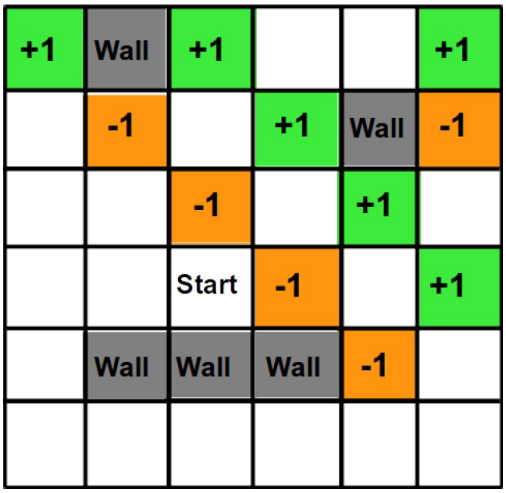

Figure 1: Maze environment as given in the assignment.

The environment provided to us is a 6×6 grid world that contains reward states, penalty states, walls, and empty cells. The grid is shown in Figure 1, where green cells correspond to reward states, brown cells correspond to penalty states, grey cells correspond to walls, and white cells correspond to empty cells.

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.

2 Setup

2.1 Grid World Representation

The maze is represented using a 2D array of size 6×6.

```
GRID = [
    ["G", "X", "G", " ", " ", "G",],
    [" ", "B", " ", "G", "X", "B",],
    [" ", " ", "B", " ", "G", " ",],
    [" ", " ", " ", "B", " ", "G",],
    [" ", "X", "X", "X", "B", " ",],
    [" ", " ", " ", " ", " ", " ",],
]
```

Listing 1: Maze environment as defined in `config.py`. "G" is used to denote green cells, "B" is used to denote brown cells, "X" is used to denote wall cells, and " " is used to denote white cells.

2.2 Rewards Mapping

The rewards are stored using a map between states, s, and their corresponding rewards, $R(s)$.

```
REWARDS = {
    "G": +1.00,
    "B": -1.00,
    " ": -0.04,
}
```

Listing 2: Rewards mapping as defined in `config.py`. No reward is stored for wall cells since it is not a valid state.

2.3 Actions

The actions are stored using a map between their string representation and the change in states they produce.

```
ACTIONS = {
    "→": (0, 1),
    "↑": (-1, 0),
    "←": (0, -1),
    "↓": (1, 0)
}
```

Listing 3: Actions mapping as defined in `config.py`. "→" represents a rightward movement, "↑" an upward movement, "←" a leftward movement, and "↓" a downward movement.

For example, "→" is mapped to $(0, 1)$, meaning that the agent desires to change its current state from index (m, n) to $(m, n + 1)$. If this new position cannot be reached, i.e., its outside the maze or a wall cell, then the agent stays in the same position.

3 MDP Base

In this section we include the implementation of the `MDPBase` class as in `mdp_base.py`.

3.1 Moving in the Grid

Since the agent can't step outside the grid or go on a wall state, we implement a basic function, `move`, to check where the agent would end up if it executes action a in state s.

```
def move(self, curr_position, action):
    x, y = curr_position; del_x, del_y = action
    new_x = x+del_x; new_y = y+del_y
    if (new_x, new_y) in self.states:
        return (new_x, new_y)
    else:
        return (x, y)
```

Listing 4: Function `move` as defined in the `MDPBase` class in `mdp_base.py`

As shown in Listing 4, if the agent runs into a wall or the edge of the grid, it stays in the same state it executed the action from.

3.2 Transition Model

The probabilistic transition model, as given in the assignment, executes the intended action with a certain probability, which is defined in `config.py`, or executes one of the two alternate actions with equal probability. We are given the probability of executing the intended outcome as 0.8, and the two alternate actions are moving in perpendicular directions with probability 0.1 for each.

For example, if the agent intends to move rightwards ($\rightarrow$: $(1, 0)$), then with 0.8 probability it will do so, and move upwards ($\uparrow$: $(0, 1)$) or downwards ($\downarrow$: $(0, -1)$) with equal probability, 0.1.

```
def probable_actions(intended_action):
    del_x, del_y = intended_action
    probs = (self.pr_intended,) + (((1_self.pr_intended)/2),) * 2
    if abs(del_x) > 0 and del_y == 0:
        return zip((intended_action, (0, +1), (0, -1)), probs)
    elif del_x == 0 and abs(del_y) > 0:
        return zip((intended_action, (+1, 0), (-1, 0)), probs)
```

Listing 5: Function `probable_actions` as defined in the `MDPBase` class in `mdp_base.py`

As shown in Listing 5, function `probable_actions` takes the intended action as a parameter and returns the intended action and alternate ones along with their execution probabilities.

3.3 Expected Utility

Expected utility of taking an action a in state s is calculated as the sum of the utilities of the states the agent can end up in weighted by the probability of such a transition.

```
def expected_utility(self, position, intended_action):
    expected_utility = 0
    for action, prob in self.probable_actions(self.actions[intended_action]):
        new_x, new_y = self.move(position, action)
        expected_utility += prob * self.utilities[new_x, new_y]
    return expected_utility
```

Listing 6: Function `expected_utility` as defined in the `MDPBase` class in `mdp_base.py`

As shown in Listing 6, function `expected_utility` returns the expected utility calculated as the sum of probability of transitioning into a state multiplied by its utility.

3.4 Optimal Policy

The optimal policy in a state is calculated as the action which maximizes the expected utility after transition.

```
def best_policy(self, position):
    best_intended_action, best_utility = None, -float("inf")
    for intended_action in self.actions:
        action_utility = self.expected_utility(position, intended_action)
        if action_utility > best_utility:
            best_intended_action = intended_action
            best_utility = action_utility
    return best_intended_action, best_utility
```

Listing 7: Function `optimal_policy` as defined in the `MDPBase` class in `mdp_base.py`

As shown in Listing 7, function `optimal_policy` finds the action that maximizes the expected utility after transition.

4 Value Iteration

Value Iteration is an algorithm used to determine the utility of each state, and use that estimate to determine the optimal policy for an agent. The method involves calculating the utility of each state in the world through multiple iterations until the values for all states converge. Using these values, we can compute the optimal policy for the agent by selecting actions that maximize the expected utility in the agent's state sequences.

The utility of any state is calculated by summing the reward assigned to that state with the expected discounted utility of the next states. We assume that the agent chooses the action that maximizes the expected utility at each state, making it a rational agent.

Algorithm 1 Value Iteration

procedure VALUE_ITERATION(mdp, ϵ)
    $U \leftarrow 0$; $U' \leftarrow 0$
    $\delta \leftarrow \infty$
    while $\delta \geq \epsilon (1 - \gamma) / \gamma$ **do**
        $U \leftarrow U'$; $\delta \leftarrow 0$
        for each state s **in** S **do**
            $U'[s] \leftarrow R(s) + \max_{a \in A(s)} \sum_{s'} P(s'|s, a) U[s']$
            if $|U'[s] - U[s]| > \delta$ **then**
                $\delta \leftarrow |U'[s] - U[s]|$
            end if
        end for
    end while
    return U
end procedure

The algorithm begins by initializing the utility of every state to be 0. It then updates the utility of each state using the Bellman Equation, which recursively relates the utility of a state to the utility of its neighboring states to find the updated value at the end of each iteration. The algorithm continues to iterate until the utilities for all states converge. Finally, the optimal policy for the grid world is calculated based on these final utilities.

4.1 Implementation

We implement the Value Iteration algorithm using the class `ValueIterator` implemented in the file `value_iteration.py`.

4.1.1 One Iteration

In each iteration of Value Iteration, we update the utility estimates using the Bellman update rule.

```
def one_iteration(self):
    U = np.zeros_like(self.utilities, dtype=float)
    for (m, n) in self.states:
        best_intended_action, best_utility = self.best_policy((m, n))
        U[m, n] = self.rewards[self.grid[m, n]] \
            + self.discount_factor * best_utility
        self.policy[m, n] = best_intended_action
    return U
```

Listing 8: Function `one_iteration` as defined in the `ValueIteration` class in `value_iteration.py`

As shown in Listing 8, we use the current utility estimates to calculate the action giving the best expected utility after transition. We then set it as the optimal policy and update the utility of the state.

4.1.2 Iterating till Convergence

We run the Value Iteration algorithm until convergence is achieved, i.e., the maximum change in utility values, δ, is lower than $\epsilon(1-\gamma)/\gamma$. We define $\epsilon = 1$ and the discount factor, $\gamma = 0.99$ in `config.py`.

```
def value_iteration(self, max_error):
    utilities = list()
    delta = float('inf')
    upper_bound = max_error*(1-self.discount_factor)/self.discount_factor
    while delta >= upper_bound:
        U = self.one_iteration()
        delta = np.max(np.abs(U-self.utilities))
        self.utilities = U
        utilities.append(self.utilities)
    return utilities
```

Listing 9: Function `value_iteration` as defined in the `ValueIteration` class in `value_iteration.py`

4.2 Results

4.2.1 Converged Utilities

99.008		94.053	92.883	91.663	92.336
97.401	94.891	93.553	93.406		89.926
95.956	94.594	92.302	92.184	92.110	90.803
94.562	93.460	92.240	90.123	90.822	90.896
93.320				88.556	89.575
91.945	90.737	89.543	88.364	87.577	88.306

Table 1: Value Iteration - Converged utility estimates

The utility values (truncated to 3 decimal places) for each state are shown in Table 1. The wall cells are empty since they cannot be reached. Clearly, the top-left cell has the highest utility since an upward action in this state keeps the agent in the same cell, and it can keep accumulating the highest reward (+1.00).

4.2.2 Optimal Policy

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

Table 2: Value Iteration - Optimal policy

The optimal policy for each state is shown in Table 2. The wall cells are empty since they cannot be reached. As discussed above, the optimal policy for the top-left cell is to move up.

4.2.3 Analysis

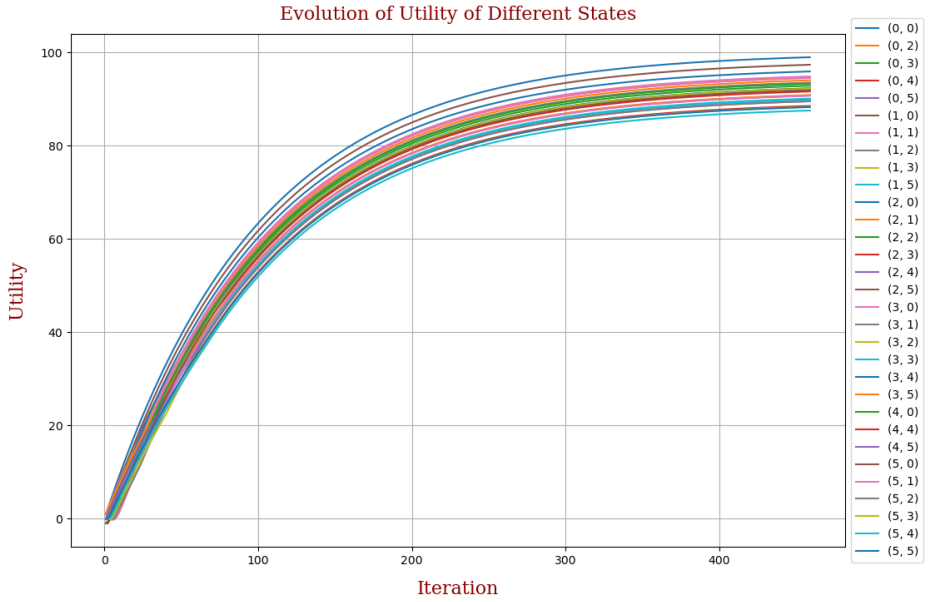

Figure 2: Evolution of utilities using Value Iteration algorithm.

We plot out the evolution of utility values for each state in Figure 2. The algorithm converges in 459 iterations, and took 0.449 seconds to converge.

5 Policy Iteration

Policy Iteration algorithm involves starting with an initial assumed policy and then calculating the optimal policy over multiple iterations. The algorithm alternates between two procedures: policy evaluation and policy improvement.

During policy evaluation, we calculate the utilities of all states taking the current policy as fixed. After applying the Bellman update some number of times, the algorithm performs a policy improvement step. In this step, using the utility values estimated to that point, the current policy is updated as the action that maximizes the expected utility for a state.

The algorithm continues to alternate between policy evaluation and policy improvement until the policy found is stable, and there are no updates in new iterations. This stable policy is the optimal policy for the grid world.

Algorithm 2 Policy Iteration

```
1: procedure POLICY_ITERATION(mdp)
2:     U ← 0; randomly initialize π
3:     changed ← true
4:     while changed do
5:         U ← POLICY_EVALUATION(π, U, mdp)
6:         changed ← false
7:         for each state s in S do
8:             if max_{a∈A(s)} Σ_{s'} P(s'|s,a) U[s'] > Σ_{s'} P(s'|s,π(s)) U[s'] then
9:                 π[s] ← argmax_{a∈A(s)} Σ_{s'} P(s'|s,a) U[s']
10:                changed ← true
11:            end if
12:        end for
13:    end while
14:    return π
15: end procedure
```

5.1 Implementation

We implement the Policy Iteration algorithm using the class `PolicyIterator` implemented in the file `policy_iteration.py`.

5.1.1 Policy Evaluation

In Policy Evaluation, we estimate the utility of each state using a simplified version of the Bellman update rule where we assume a fixed policy:

$$U[s] = R(s) + \sum_{s'} P(s'|s, \pi(s))\, U[s']$$

```
def policy_evaluation(self):
    for _ in range(self.evaluation_iterations):
        U_pi = np.zeros_like(self.utilities, dtype=float)
        for (m, n) in self.states:
            U_pi[m, n] = self.rewards[self.grid[m, n]] + \
                self.discount_factor * self.expected_utility((m, n), self.policy[m, n])
        self.utilities = U_pi
```

Listing 10: Function `policy_evaluation` as defined in the `PolicyIteration` class in `policy_iteration.py`

This step is implemented in Listing 10. For utility estimation, the function uses a fixed number of steps, which is passed in the constructor. We have defined this constant as 100 in `config.py`.

5.1.2 One Iteration

In each iteration of Policy Iteration, we first do policy evaluation, followed by policy improvement, wherein we find the best action for each state considering the current utility values.

```
def one_iteration(self):
    self.policy_evaluation()
    converged = True
    for (m, n) in self.states:
        best_intended_action, best_utility = self.best_policy((m, n))
        if best_utility > self.expected_utility((m, n), self.policy[m, n]):
            self.policy[m, n] = best_intended_action
            converged = False
    return converged
```

Listing 11: Function `one_iteration` as defined in the `PolicyIteration` class in `policy_iteration.py`

As shown in Listing 11, we first perform policy evaluation, following which we use the current utility estimates to calculate the action giving the best expected utility after transition for each state. We then set then as the optimal policy.

5.1.3 Iterating till Convergence

We run the Policy Iteration algorithm until the optimal policy converges, i.e., the optimal policy does not change for any state.

```
def policy_iteration(self):
    utilities, converged = list(), False
    while not converged:
        converged = self.one_iteration()
        utilities.append(self.utilities)
    return utilities
```

Listing 12: Function `policy_iteration` as defined in the `PolicyIteration` class in `policy_iteration.py`

As show in Listing 12, we keep updating the optimal policy until convergence is achieved. The function returns the sequence of utility estimates from each iteration.

5.2 Results

5.2.1 Converged Utilities

99.482		94.527	93.357	92.137	92.810
97.875	95.365	94.027	93.880		90.400
96.430	95.068	92.776	92.658	92.584	91.277
95.036	93.934	92.714	90.597	91.296	91.370
93.794				89.030	90.049
92.419	91.211	90.017	88.838	88.051	88.780

Table 3: Policy Iteration - Converged utility estimates

The utility values (truncated to 3 decimal places) for each state are shown in Table 3. The wall cells are empty since they cannot be reached. Clearly, the top-left cell has the highest utility since an upward action in this state keeps the agent in the same cell, and it can keep accumulating the highest reward (+1.00).

5.2.2 Optimal Policy

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

Table 4: Policy Iteration - Optimal policy

The optimal policy for each state is shown in Table 4. The wall cells are empty since they cannot be reached. As discussed above, the optimal policy for the top-left cell is to move up. It is interesting to note that this policy is the same as the one given by Value Iteration.

These results are stored in `part1-assets/policy-iteration-results.txt`.

5.2.3 Analysis

We plot out the evolution of utility values for each state in Figure 3. The algorithm converges in 6 iterations, and took 0.167 seconds to converge on the given maze, making it approximately 2.7 times faster than Value Iteration. These values are dependant on the constants used for the algorithm, but the optimal policy at convergence should remain the same. The number of iterations required for convergence is also dependant on the constants chosen.

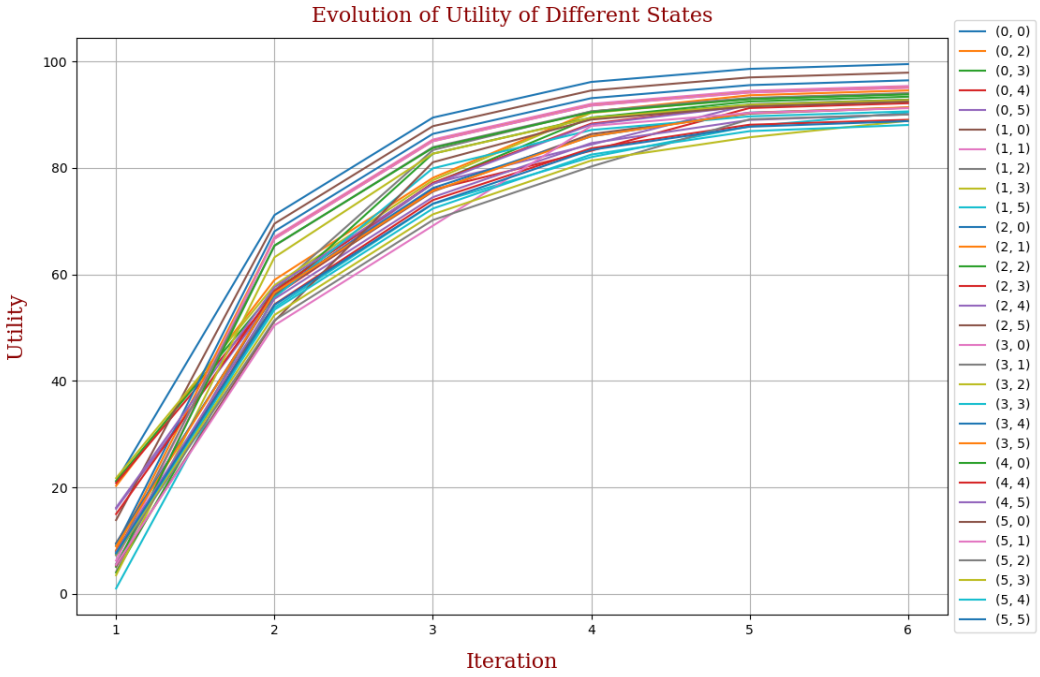

Figure 3: Evolution of utilities using Policy Iteration algorithm.

6 Bonus Question

For Part 2 of the assignment, we generated random grids of size $n \times n$, where $n \in \{10, 20, \ldots, 100\}$. The cell states are sampled independently and identically from a distribution with mass function estimated using the distribution of the states in the question.

```
states = ("G", "B", " ", "X")
probs = (1/6, 1/6, 1/2, 1/6)
shape = (args.nrows, args.ncols)
GRID = np.random.choice(a=states, size=shape, p=probs)
```

Listing 13: Code block for generating random grids of arbitrary size, as defined in `part2.py`

The results are stored in the directory `part2-assets`, where each subdirectory corresponds to a different size of the grid. We use these runs to analyze how the number of iterations required for convergence and the corresponding running time of Value Iteration and Policy Iteration algorithms scale with the size of the grid.

6.1 Number of Iterations

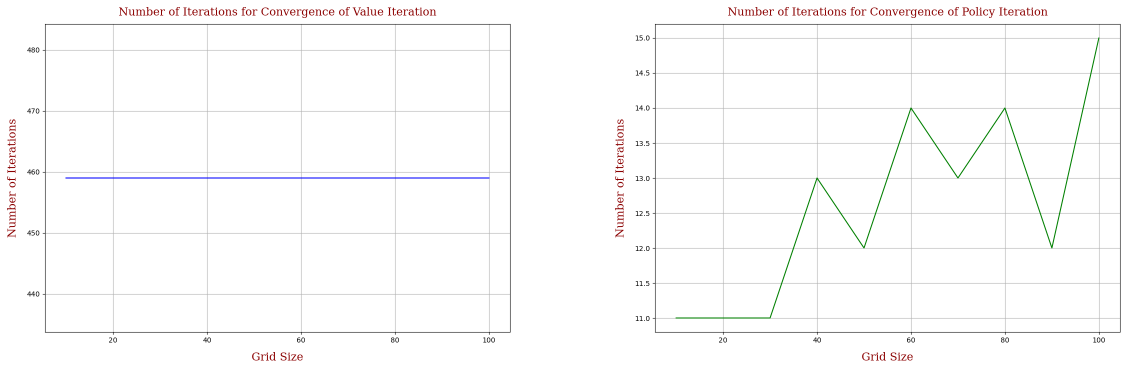

Figure 4: Number of iterations versus grid size. Top: Value Iteration. Bottom: Policy Iteration.

Interestingly, the number of iterations of Value Iteration required was exactly the same for all grid sizes - 459 iterations. The number of iterations with Policy Iteration had a slight trend but not significant enough. The numbers of iterations required for convergence of the algorithms are included in Figure 4.

6.2 Running Time

The running time of the algorithms for convergence shows a super-linear trend. By and large, Value Iteration seems to be the more expensive of the two algorithms, but only by a small margin. The running times for both algorithms are included in Figure 5.

Caution: We understand that a single run with a randomly generated grid may give noisy results, but, due to computational constraints, we restricted ourselves to one run per grid size. Regardless, we do see some interesting results.

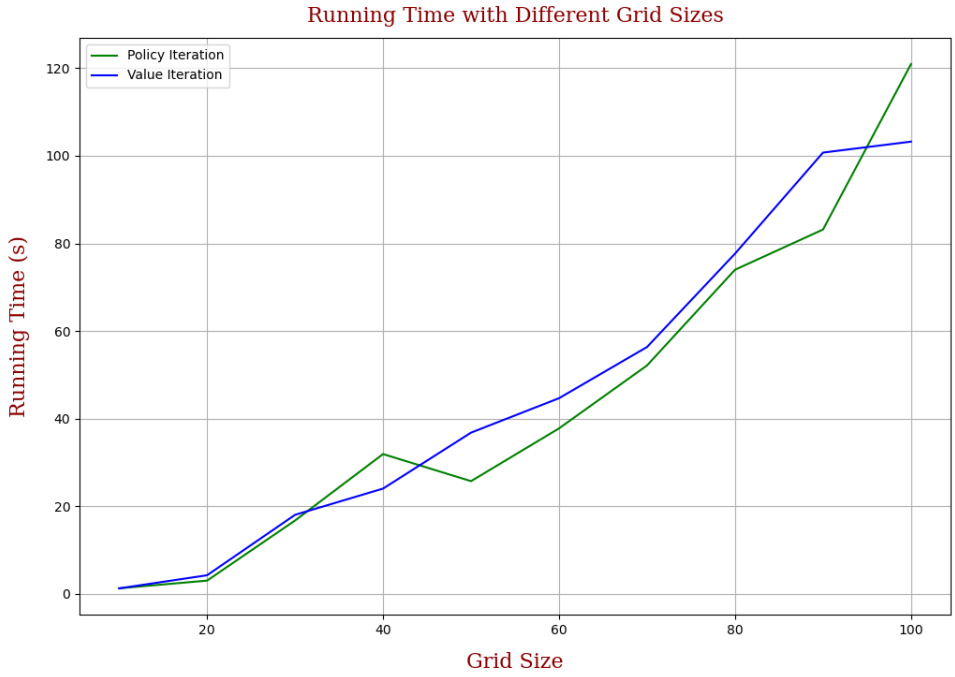

Figure 5: Running time as a function of the grid size.

7 Steps to Reproduce

7.1 Directory Contents

- `part1-assets/`: contains results for part 1
- `part2-assets/`: contains results for part 2
- `Assignment1-Instructions.pdf`: instructions for the assignment
- `Assignment1-Submission.pdf`: submission report
- `config.py`: constants and configurations to be used
- `mdp_base.py`: base class for value iterator and policy iterator classes
- `value_iteration.py`: defines value iterator class
- `policy_iteration.py`: defines policy iterator class
- `part1.py`: driver code for part 1
- `part2.py`: driver code for part 2
- `utils.py`: utility function for logging results

7.2 Part 1

To reproduce the results of part 1 of the assignment, execute

```
python part1.py
```

You can set the parameters for the run in `config.py`. The results are saved in the directory `part1-assets`:

- `value-iteration-plot.png`: the plot of evolution of utility values in the Value Iteration run
- `value-iteration-results.png`: text file containing the number of iterations required for convergence and the converged utility estimates and the optimal policy in the Value Iteration run
- `policy-iteration-plot.png`: the plot of evolution of utility values in the Policy Iteration run
- `policy-iteration-results.png`: text file containing the number of iterations required for convergence and the converged utility estimates and the optimal policy in the Policy Iteration run

7.3 Part 2

To reproduce the results of part 2 of the assignment, execute

```
python part2.py --nrows 100 --ncols 100
```

You can set the parameters for the run in `config.py`. The size of the grid can be controlled from the command line using `nrows` and `ncols` flags. The results are saved in the directory `part2-assets/{nrows}x{ncols}-grid`:

- `value-iteration-plot.png`: the plot of evolution of utility values of 30 randomly selected states in the Value Iteration run
- `value-iteration-results.png`: text file containing the number of iterations required for convergence and the running time of the algorithm, along with the converged utility estimates and the optimal policy in the Value Iteration run
- `policy-iteration-plot.png`: the plot of evolution of utility values of 30 randomly selected states in the Policy Iteration run
- `policy-iteration-results.png`: text file containing the number of iterations required for convergence and the running time of the algorithm, along with the converged utility estimates and the optimal policy in the Policy Iteration run