Assignment 1

BY YUSEN ZHENG

March 13, 2023

Email: zys0794@sjtu.edu.cn

Student ID: 520021911173

1 Introduction

In this report, we build a simple Gridworld. We implement **iterative policy evaluation**, **policy iteration** and **value iteration** methods to find the **optimal policy** and **optimal value** for each grid. The gridworld is shown in the figure below.

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23
24	25	26	27	28	29
30	31	32	33	34	35

2 Gridworld Environment

We define **the grid class**, which contains information such as the position of the grid in the entire gridworld, value, policy, and whether it is a terminal state.

```
class Grid:
    def __init__(self, position, value=.0, is_terminal=False):
        self.val = value
        self.pos = position
        self.act = {'n': .0, 'e': .0, 's': .0, 'w': .0} if is_terminal else {
            'n': 0.25, 'e': 0.25, 's': 0.25, 'w': 0.25}
        self.is_terminal = is_terminal
```

We define **the gridworld class**, which contains information such as the length and width of the grid world, the position of the terminal state, the discount factor (=1), the reward (= -1), and the threshold (the default value is set to 0.0001) of the iterative algorithm.

```
class GridWorld:
    def __init__(self, width, height, terminal_list, gamma, theta=.0001):
        self.w = width
        self.h = height
        self.terminal_list = terminal_list
        self.gamma = gamma
        self.theta = theta
        self.r = -1
        self.grid_list = []
        for i in range(width*height):
            self.grid_list.append(Grid(i, is_terminal=i in terminal_list))
```

The gridworld in the assignment can be constructed by `World = GridWorld(6, 6, [1,35], 1.0)`.

In the iterative process, $\sum_{s',r} p(s',r|\, s,a)[r+\gamma V(s')]$ needs to be calculated multiple times, so this calculation process is encapsulated into a function.

```
    def act_val(self, pos, action):
        grid = self.grid_list[pos]
        val = {
            "n": self.r + self.gamma *
            (self.grid_list[pos-self.w].val if pos >=
             self.w else grid.val),
            "e": self.r + self.gamma *
            (self.grid_list[pos+1].val if (pos+1) %
             self.w else grid.val),
            "s": self.r + self.gamma *
            (self.grid_list[pos+self.w].val if (pos +
                                                 self.w) < self.w*self.h else
grid.val),
            "w": self.r + self.gamma *
            (self.grid_list[pos-1].val if pos % self.w else grid.val)
        }
        return val.get(action, lambda: "Invalid action")
```

We also implemented the function of formatting and printing the value and policy of each grid point in the grid world, the code (function: `print_val` and `print_policy`) is in the appendix.

3 Iterative Policy Evaluation

3.1 Algorithm

Iterative policy evaluation is based on Bellman expectation equation.

$$v_{k+1}(s) = \sum_{a\in\mathcal{A}} \pi(a|\, s)\left(\mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_k(s')\right)$$

The algorithm is as follows:

Iterative Policy Evaluation, for estimating $V \approx v_\pi$

Input π, the policy to be evaluated
Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
 $\Delta \leftarrow 0$
 Loop for each $s \in \mathcal{S}$:
  $v \leftarrow V(s)$
  $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s',r\,|\,s,a)\big[r+\gamma V(s')\big]$
  $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

3.2 Implement

```
def eval(self):
    while True:
```

```
        delta = .0
        update_val = []
        for i in range(self.w*self.h):
            grid = self.grid_list[i]
            if grid.is_terminal:
                update_val.append(.0)
            else:
                v = grid.val
                val = grid.act['n']*self.act_val(i, 'n')+grid.act['e'] * \
                    self.act_val(i,
                                'e')+grid.act['s']*self.act_val(i,
's')+grid.act['w']*self.act_val(i, 'w')
                delta = max(delta, abs(val-v))
                update_val.append(val)
        for i in range(self.w*self.h):
            self.grid_list[i].val = update_val[i]
        if delta < self.theta:
            for i in range(self.w*self.h):
                grid = self.grid_list[i]
                if grid.is_terminal:
                    continue
                for action in ['n', 'e', 's', 'w']:
                    grid.act[action] = .0
                else:
                    act_val_lst = [self.act_val(i, 'n'), self.act_val(
                        i, 'e'), self.act_val(i, 's'), self.act_val(i, 'w')]
                    value = max(act_val_lst)
                    act = [idx for idx, val in enumerate(
                        act_val_lst) if val == value]
                    pr = 1/len(act)
                    for action in [list(grid.act)[j] for j in act]:
                        grid.act[action] = pr
            break
```

3.3 Result

If agent follows uniform random policy $\pi(n|\cdot)=\pi(s|\cdot)=\pi(w|\cdot)=\pi(e|\cdot)=0.25$, the value function $v_\infty(s)$ and greedy policy w.r.t. $v_\infty(s)$ are as shown in the figure below.

```
Iterative Policy Evaluation
-18.17  0.00    -29.22  -44.06  -51.55  -54.68
-32.34  -30.17  -39.59  -47.41  -51.93  -53.80
-44.68  -44.73  -47.58  -50.05  -50.95  -50.79
-52.96  -52.50  -51.95  -50.26  -47.05  -43.61
-57.71  -56.38  -53.44  -48.01  -39.37  -29.00
-59.78  -57.86  -53.42  -44.96  -29.44  0.00
→               ←       ←       ←       ←
↑       ↑       ↑       ←       ←       ↓
↑       ↑       ↑       ↑       ↓       ↓
↑       ↑       ↑       →       ↓       ↓
↑       ↑       →       →       →       ↓
↑       →       →       →       →
```

4 Policy Iteration

4.1 Algorithm

The algorithm is as follows:

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Loop:
   $\quad \Delta \leftarrow 0$
   $\quad$ Loop for each $s \in \mathcal{S}$:
   $\quad\quad v \leftarrow V(s)$
   $\quad\quad V(s) \leftarrow \sum_{s',r} p(s',r\,|\,s,\pi(s))\big[r + \gamma V(s')\big]$
   $\quad\quad \Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   $\quad$ *old-action* $\leftarrow \pi(s)$
   $\quad \pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s',r\,|\,s,a)\big[r + \gamma V(s')\big]$
   $\quad$ If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

4.2 Implement

```
def policy_iter(self):
    while True:
        while True:
            delta = 0
            update_val = []
            for i in range(self.w*self.h):
                grid = self.grid_list[i]
                if grid.is_terminal:
                    update_val.append(.0)
                else:
                    v = grid.val
                    val = grid.act['n']*self.act_val(i, 'n')+grid.act['e'] * \
                        self.act_val(i,
                                'e')+grid.act['s']*self.act_val(i,
's')+grid.act['w']*self.act_val(i, 'w')
                    delta = max(delta, abs(val-v))
                    update_val.append(val)
            for i in range(self.w*self.h):
                self.grid_list[i].val = update_val[i]
            if delta < self.theta:
                break
```

```
is_stable = True
for i in range(self.w*self.h):
    grid = self.grid_list[i]
    if grid.is_terminal:
        continue
    old_action = grid.act.copy()
    for action in ['n', 'e', 's', 'w']:
        grid.act[action] = .0
    act_val_lst = [self.act_val(i, 'n'), self.act_val(
        i, 'e'), self.act_val(i, 's'), self.act_val(i, 'w')]
    act = [idx for idx, val in enumerate(
        act_val_lst) if val == max(act_val_lst)]
    pr = 1/len(act)
    for action in [list(grid.act)[j] for j in act]:
        grid.act[action] = pr
    if grid.act != old_action:
        is_stable = False
if is_stable:
    break
```

4.3 Result

Through the policy iteration algorithm implemented above, we get the optimal value function and policy function of each state, as follows

```
Policy Iteration
-1.00   0.00    -1.00   -2.00   -3.00   -4.00
-2.00   -1.00   -2.00   -3.00   -4.00   -4.00
-3.00   -2.00   -3.00   -4.00   -4.00   -3.00
-4.00   -3.00   -4.00   -4.00   -3.00   -2.00
-5.00   -4.00   -4.00   -3.00   -2.00   -1.00
-5.00   -4.00   -3.00   -2.00   -1.00   0.00
→               ←       ←       ←       ←
↑→      ↑       ↑←      ↑←      ↑←      ↓
↑→      ↑       ↑←      ↑←      →↓      ↓
↑→      ↑       ↑←      →↓      →↓      ↓
↑→      ↑       →↓      →↓      →↓      ↓
→       →       →       →       →
```

5 Value Iteration

5.1 Algorithm

Value Iteration is based on the equation:

$$v_{k+1}(s) = \max_{a\in\mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right)$$

The algorithm is as follows:

Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
| $\Delta \leftarrow 0$
| Loop for each $s \in \mathcal{S}$:
| $v \leftarrow V(s)$
| $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
| $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$

5.2 Implement

```
def value_iter(self):
    while True:
        delta = 0
        update_val = []
        for i in range(self.w*self.h):
            grid = self.grid_list[i]
            if grid.is_terminal:
                update_val.append(.0)
            else:
                v = grid.val
                act_val_lst = [self.act_val(i, 'n'), self.act_val(
                    i, 'e'), self.act_val(i, 's'), self.act_val(i, 'w')]
                max_val = max(act_val_lst)
                update_val.append(max_val)
                delta = max(delta, abs(v-max_val))
        for i in range(self.w*self.h):
            self.grid_list[i].val = update_val[i]
        if delta < self.theta:
            break
    for i in range(self.w*self.h):
        grid = self.grid_list[i]
        if grid.is_terminal:
            continue
        for action in ['n', 'e', 's', 'w']:
            grid.act[action] = .0
        act_val_lst = [self.act_val(i, 'n'), self.act_val(
            i, 'e'), self.act_val(i, 's'), self.act_val(i, 'w')]
        act = [idx for idx, val in enumerate(
            act_val_lst) if val == max(act_val_lst)]
        pr = 1/len(act)
        for action in [list(grid.act)[j] for j in act]:
            grid.act[action] = pr
```

5.3 Result

Through the value iteration algorithm implemented above, we get the optimal value function and policy function of each state, as follows. We find that this is the **same** convergence result as the policy iteration algorithm. On the other hand, during the experiments, we found that the value iteration converges **faster** than the policy iteration.

```
Value Iteration
-1.00   0.00    -1.00   -2.00   -3.00   -4.00
-2.00   -1.00   -2.00   -3.00   -4.00   -4.00
-3.00   -2.00   -3.00   -4.00   -4.00   -3.00
-4.00   -3.00   -4.00   -4.00   -3.00   -2.00
-5.00   -4.00   -4.00   -3.00   -2.00   -1.00
-5.00   -4.00   -3.00   -2.00   -1.00   0.00
→               ←       ←       ←       ←
↑→      ↑       ↑←      ↑←      ↑←      ↓
↑→      ↑       ↑←      ↑←      →↓      ↓
↑→      ↑       ↑←      →↓      →↓      ↓
↑→      ↑       →↓      →↓      →↓      ↓
→       →       →       →       →
```

A Source Code

```
class Grid:
    def __init__(self, position, value=.0, is_terminal=False):
        self.val = value
        self.pos = position
        self.act = {'n': .0, 'e': .0, 's': .0, 'w': .0} if is_terminal else {
            'n': 0.25, 'e': 0.25, 's': 0.25, 'w': 0.25}
        self.is_terminal = is_terminal


class GridWorld:
    def __init__(self, width, height, terminal_list, gamma, theta=.0001):
        self.w = width
        self.h = height
        self.terminal_list = terminal_list
        self.gamma = gamma
        self.theta = theta
        self.r = -1
        self.grid_list = []
        for i in range(width*height):
            self.grid_list.append(Grid(i, is_terminal=i in terminal_list))

    def print_val(self):
        for i in range(self.w*self.h):
            if i % self.w == 0 and i != 0:
                print(f'\n{self.grid_list[i].val:.2f}\t', end='')
            elif i != self.w*self.h-1:
                print(f'{self.grid_list[i].val:.2f}\t', end='')
            else:
                print(f'{self.grid_list[i].val:.2f}\t')

    def print_policy(self):
        arrows = ['↑', '→', '↓', '←']
```

```
        for i in range(self.w*self.h):
            mov = ''
            for j in range(4):
                if list(self.grid_list[i].act.items())[j][1] != .0:
                    mov += arrows[j]
            if i % self.w == 0 and i != 0:
                print(f'\n{mov}\t', end='')
            elif i != self.w*self.h-1:
                print(f'{mov}\t', end='')
            else:
                print(f'{mov}\t')

    def act_val(self, pos, action):
        grid = self.grid_list[pos]
        val = {
            "n": self.r + self.gamma *
            (self.grid_list[pos-self.w].val if pos >=
             self.w else grid.val),
            "e": self.r + self.gamma *
            (self.grid_list[pos+1].val if (pos+1) %
             self.w else grid.val),
            "s": self.r + self.gamma *
            (self.grid_list[pos+self.w].val if (pos +
                                                self.w) < self.w*self.h else
grid.val),
            "w": self.r + self.gamma *
            (self.grid_list[pos-1].val if pos % self.w else grid.val)
        }
        return val.get(action, lambda: "Invalid action")

    def eval(self):
        while True:
            delta = .0
            update_val = []
            for i in range(self.w*self.h):
                grid = self.grid_list[i]
                if grid.is_terminal:
                    update_val.append(.0)
                else:
                    v = grid.val
                    val = grid.act['n']*self.act_val(i, 'n')+grid.act['e'] * \
                        self.act_val(i,
                                     'e')+grid.act['s']*self.act_val(i,
's')+grid.act['w']*self.act_val(i, 'w')
                    delta = max(delta, abs(val-v))
                    update_val.append(val)
            for i in range(self.w*self.h):
                self.grid_list[i].val = update_val[i]
            if delta < self.theta:
                for i in range(self.w*self.h):
                    grid = self.grid_list[i]
                    if grid.is_terminal:
                        continue
                    for action in ['n', 'e', 's', 'w']:
                        grid.act[action] = .0
                    else:
```

```
                        act_val_lst = [self.act_val(i, 'n'), self.act_val(
                            i, 'e'), self.act_val(i, 's'), self.act_val(i,
'w')]
                        value = max(act_val_lst)
                        act = [idx for idx, val in enumerate(
                            act_val_lst) if val == value]
                        pr = 1/len(act)
                        for action in [list(grid.act)[j] for j in act]:
                            grid.act[action] = pr
                break

    def policy_iter(self):
        while True:
            while True:
                delta = 0
                update_val = []
                for i in range(self.w*self.h):
                    grid = self.grid_list[i]
                    if grid.is_terminal:
                        update_val.append(.0)
                    else:
                        v = grid.val
                        val = grid.act['n']*self.act_val(i, 'n')+grid.act['e']
* \
                            self.act_val(i,
                                        'e')+grid.act['s']*self.act_val(i,
's')+grid.act['w']*self.act_val(i, 'w')
                        delta = max(delta, abs(val-v))
                        update_val.append(val)
                for i in range(self.w*self.h):
                    self.grid_list[i].val = update_val[i]
                if delta < self.theta:
                    break
            is_stable = True
            for i in range(self.w*self.h):
                grid = self.grid_list[i]
                if grid.is_terminal:
                    continue
                old_action = grid.act.copy()
                for action in ['n', 'e', 's', 'w']:
                    grid.act[action] = .0
                act_val_lst = [self.act_val(i, 'n'), self.act_val(
                    i, 'e'), self.act_val(i, 's'), self.act_val(i, 'w')]
                act = [idx for idx, val in enumerate(
                    act_val_lst) if val == max(act_val_lst)]
                pr = 1/len(act)
                for action in [list(grid.act)[j] for j in act]:
                    grid.act[action] = pr
                if grid.act != old_action:
                    is_stable = False
            if is_stable:
                break

    def value_iter(self):
        while True:
            delta = 0
```

```
            update_val = []
            for i in range(self.w*self.h):
                grid = self.grid_list[i]
                if grid.is_terminal:
                    update_val.append(.0)
                else:
                    v = grid.val
                    act_val_lst = [self.act_val(i, 'n'), self.act_val(
                        i, 'e'), self.act_val(i, 's'), self.act_val(i, 'w')]
                    max_val = max(act_val_lst)
                    update_val.append(max_val)
                    delta = max(delta, abs(v-max_val))
            for i in range(self.w*self.h):
                self.grid_list[i].val = update_val[i]
            if delta < self.theta:
                break
        for i in range(self.w*self.h):
            grid = self.grid_list[i]
            if grid.is_terminal:
                continue
            for action in ['n', 'e', 's', 'w']:
                grid.act[action] = .0
            act_val_lst = [self.act_val(i, 'n'), self.act_val(
                i, 'e'), self.act_val(i, 's'), self.act_val(i, 'w')]
            act = [idx for idx, val in enumerate(
                act_val_lst) if val == max(act_val_lst)]
            pr = 1/len(act)
            for action in [list(grid.act)[j] for j in act]:
                grid.act[action] = pr


if __name__ == '__main__':
    World1 = GridWorld(6, 6, [1,35], 1.0)
    World1.eval()
    print('Iterative Policy Evaluation')
    World1.print_val()
    World1.print_policy()
    World2 = GridWorld(6, 6, [1, 35], 1.0)
    World2.policy_iter()
    print('\nPolicy Iteration')
    World2.print_val()
    World2.print_policy()
    World3 = GridWorld(6, 6, [1, 35], 1.0)
    World3.value_iter()
    print('\nValue Iteration')
    World3.print_val()
    World3.print_policy()
```