A1_DynProg

February 5, 2021

Student Name: Vijaykumar Maraviya

Student Number: 1006040320

0.0.1 Question: What is the purpose of OpenAI gyms and how is it going to help us in our RL education?

According to OpenAI gym documentation, the main objectives of gym is as follows:

1) Provide standard set of environments that are easy to setup and use.
2) The goal is to have better benchmarks to compare the outcome of research through standardization of problem defination such as reward function or the set of actions.

In education, it will let researchers and students focus on learning and development of RL algorithms, rather than spending time on problem defination and setup. It will also make dissemination of knowledge convenient.

[1]:
```
import gym
import numpy as np
```

[2]:
```
gym.envs.register(
    id='FrozenLakeNotSlippery-v0',
    entry_point='gym.envs.toy_text:FrozenLakeEnv',
    kwargs={'map_name' : '4x4', 'is_slippery': False},
    max_episode_steps=100,
    reward_threshold=0.74
)
```

[3]:
```
# Create the gridworld-like environment
env=gym.make('FrozenLakeNotSlippery-v0')
# Let's look at the model of the environment (i.e., P):
env.env.P
# presentation of P
```

[3]:
```
{0: {0: [(1.0, 0, 0.0, False)],
  1: [(1.0, 4, 0.0, False)],
  2: [(1.0, 1, 0.0, False)],
  3: [(1.0, 0, 0.0, False)]},
 1: {0: [(1.0, 0, 0.0, False)],
```

```
 1: [(1.0, 5, 0.0, True)],
 2: [(1.0, 2, 0.0, False)],
 3: [(1.0, 1, 0.0, False)]},
2: {0: [(1.0, 1, 0.0, False)],
 1: [(1.0, 6, 0.0, False)],
 2: [(1.0, 3, 0.0, False)],
 3: [(1.0, 2, 0.0, False)]},
3: {0: [(1.0, 2, 0.0, False)],
 1: [(1.0, 7, 0.0, True)],
 2: [(1.0, 3, 0.0, False)],
 3: [(1.0, 3, 0.0, False)]},
4: {0: [(1.0, 4, 0.0, False)],
 1: [(1.0, 8, 0.0, False)],
 2: [(1.0, 5, 0.0, True)],
 3: [(1.0, 0, 0.0, False)]},
5: {0: [(1.0, 5, 0, True)],
 1: [(1.0, 5, 0, True)],
 2: [(1.0, 5, 0, True)],
 3: [(1.0, 5, 0, True)]},
6: {0: [(1.0, 5, 0.0, True)],
 1: [(1.0, 10, 0.0, False)],
 2: [(1.0, 7, 0.0, True)],
 3: [(1.0, 2, 0.0, False)]},
7: {0: [(1.0, 7, 0, True)],
 1: [(1.0, 7, 0, True)],
 2: [(1.0, 7, 0, True)],
 3: [(1.0, 7, 0, True)]},
8: {0: [(1.0, 8, 0.0, False)],
 1: [(1.0, 12, 0.0, True)],
 2: [(1.0, 9, 0.0, False)],
 3: [(1.0, 4, 0.0, False)]},
9: {0: [(1.0, 8, 0.0, False)],
 1: [(1.0, 13, 0.0, False)],
 2: [(1.0, 10, 0.0, False)],
 3: [(1.0, 5, 0.0, True)]},
10: {0: [(1.0, 9, 0.0, False)],
 1: [(1.0, 14, 0.0, False)],
 2: [(1.0, 11, 0.0, True)],
 3: [(1.0, 6, 0.0, False)]},
11: {0: [(1.0, 11, 0, True)],
 1: [(1.0, 11, 0, True)],
 2: [(1.0, 11, 0, True)],
 3: [(1.0, 11, 0, True)]},
12: {0: [(1.0, 12, 0, True)],
 1: [(1.0, 12, 0, True)],
 2: [(1.0, 12, 0, True)],
 3: [(1.0, 12, 0, True)]},
```

```
 13: {0: [(1.0, 12, 0.0, True)],
  1: [(1.0, 13, 0.0, False)],
  2: [(1.0, 14, 0.0, False)],
  3: [(1.0, 9, 0.0, False)]},
 14: {0: [(1.0, 13, 0.0, False)],
  1: [(1.0, 14, 0.0, False)],
  2: [(1.0, 15, 1.0, True)],
  3: [(1.0, 10, 0.0, False)]},
 15: {0: [(1.0, 15, 0, True)],
  1: [(1.0, 15, 0, True)],
  2: [(1.0, 15, 0, True)],
  3: [(1.0, 15, 0, True)]}}
```

0.0.2 Question: what is the data in this structure saying? Relate this to the course

P contains the dictionary where each key is a state and value is a dictionary. The keys of this inner dictionary are possible actions for that state. Further, each action stores a list of tuples. The first element of the tuple is transition probability, second element is a next state after transition, third element is a reward, and the fourth (last) element is True/False value to indicate if agent is done (reached terminal state). The P describes the environment model.

[4]:
```
# Now let's investigate the observation space (i.e., S using our nomenclature),
# and confirm we see it is a discrete space with 16 locations
print(env.observation_space)
```

```
Discrete(16)
```

[5]:
```
stateSpaceSize = env.observation_space.n
print(stateSpaceSize)
```

```
16
```

[6]:
```
# Now let's investigate the action space (i.e., A) for the agent->environment
# channel
print(env.action_space)
```

```
Discrete(4)
```

[7]:
```
# The gym environment has ...sample() functions that allow us to sample
# from the above spaces:
for g in range(1,10,1):
  print("sample from S:",env.observation_space.sample()," ... ","sample from A:
 ↪",env.action_space.sample())
```

```
sample from S: 15  …  sample from A: 3
sample from S: 4  …  sample from A: 1
sample from S: 13  …  sample from A: 1
sample from S: 2  …  sample from A: 3
```

```
sample from S: 5  …  sample from A: 0
sample from S: 8  …  sample from A: 3
sample from S: 13  …  sample from A: 2
sample from S: 8  …  sample from A: 2
sample from S: 1  …  sample from A: 0
```

[8]:
```
# The enviroment also provides a helper to render (visualize) the environment
env.reset()
env.render()
```

```
SFFF
FHFH
FFFH
HFFG
```

[9]:
```
# We can act as the agent, by selecting actions and stepping the environment
# through time to see its responses to our actions
env.reset()
exitCommand=False
while not(exitCommand):
    env.render()
    print("Enter the action as an integer from 0 to",env.action_space.n -1,"␣
 ↪(or exit): ")
    userInput=input()
    if userInput=="exit":
        break
    action=int(userInput)
    (observation, reward, compute, probability) = env.step(action)
    print("--> The result of taking action",action,"is:")
    print("     S=",observation)
    print("     R=",reward)
    print("     p=",probability)

env.render()
```

```
SFFF
FHFH
FFFH
HFFG
Enter the action as an integer from 0 to 3  (or exit):
1
--> The result of taking action 1 is:
     S= 4
     R= 0.0
     p= {'prob': 1.0}
  (Down)
```

```
SFFF
FHFH
FFFH
HFFG
Enter the action as an integer from 0 to 3  (or exit):
1
--> The result of taking action 1 is:
     S= 8
     R= 0.0
     p= {'prob': 1.0}
  (Down)
SFFF
FHFH
FFFH
HFFG
Enter the action as an integer from 0 to 3  (or exit):
2
--> The result of taking action 2 is:
     S= 9
     R= 0.0
     p= {'prob': 1.0}
  (Right)
SFFF
FHFH
FFFH
HFFG
Enter the action as an integer from 0 to 3  (or exit):
1
--> The result of taking action 1 is:
     S= 13
     R= 0.0
     p= {'prob': 1.0}
  (Down)
SFFF
FHFH
FFFH
HFFG
Enter the action as an integer from 0 to 3  (or exit):
2
--> The result of taking action 2 is:
     S= 14
     R= 0.0
     p= {'prob': 1.0}
  (Right)
SFFF
FHFH
FFFH
HFFG
```

```
Enter the action as an integer from 0 to 3  (or exit):
2
--> The result of taking action 2 is:
     S= 15
     R= 1.0
     p= {'prob': 1.0}
  (Right)
SFFF
FHFH
FFFH
HFFG
Enter the action as an integer from 0 to 3  (or exit):
exit
  (Right)
SFFF
FHFH
FFFH
HFFG
```

0.0.3 Question: draw a table indicating the correspondence between the action you input (a number) and the logic action performed.

action	logic
0	Move Left
1	Move Down
2	Move Right
3	Move Up

0.0.4 Question: draw a table that illustrates what the symbols on the render image mean?

symbol	Meaning
S	Start (Safe)
F	Frozen Surface (Safe)
H	Hole (fall to doom)
G	Goal

Episode ends when agent reaches the goal or fall in a Hole. It receives the reward of 1 for reaching the goal and 0 otherwise.

0.0.5 Question: Explain what the objective of the agent is in this environment?

The objective of the agent is to reach to a goal tile in a minimum number of steps, while avoiding the holes (that lead to falling in water).

0.0.6 Practical: Code up an AI that will employ random action selection in order to drive the agent. Test this random action selection agent with the above environment (i.e., code up a loop as I did above, but instead of taking input from a human user, take it from the AI you coded).

[10]:
```
# AI player with random action selection
class randomAI:
    def __init__(self):
        pass

    def select_action(self):
        return np.random.choice(4)
```

[11]:
```
# random action selection agent
agent = randomAI()
env.reset()
done=False
t = 0
while not(done):
    env.render()
    action = agent.select_action()
    (observation, reward, done, probability) = env.step(action)
    print(f"--> The agent took action {action}, moved to state S={observation},␣
 ↪and received reward {reward}")
    t = t + 1
    if done:
        env.render()
        print(f"Episode finished after {t} timesteps")
        break
```

```
SFFF
FHFH
FFFH
HFFG
--> The agent took action 0, moved to state S=0, and received reward 0.0
  (Left)
SFFF
FHFH
FFFH
HFFG
--> The agent took action 2, moved to state S=1, and received reward 0.0
  (Right)
SFFF
FHFH
FFFH
HFFG
--> The agent took action 3, moved to state S=1, and received reward 0.0
```

```
  (Up)
SFFF
FHFH
FFFH
HFFG
--> The agent took action 1, moved to state S=5, and received reward 0.0
  (Down)
SFFF
FHFH
FFFH
HFFG
Episode finished after 4 timesteps
```

0.1 Now towards dynamic programming. Note that env.env.P has the model of the environment.

0.1.1 Question: How would you represent the agent's policy function and value function?

I would represent the agent's policy and value functions using dictionaries (tables). The policy function (dictionary) will map state (key) to action (value). The value function (dictionary) will map state (key) to real number (value) representing the lucrativeness of being in that state.

0.1.2 Practical: revise the above AI solver to use a policy function in which you code the random action selections in the policy function. Test this.

[39]:
```python
def random_policy(env):
    policy = dict()
    for s in range(env.nS):
        policy[s] = np.random.choice(env.nA)
    return policy

# AI agent: action selection using random policy
class randomAI:
    def __init__(self, env):
        self.policy = random_policy(env)

    def act(self, observation):
        return self.policy[observation]
```

[40]:
```python
agent = randomAI(env)
observation = env.reset()
done=False
t = 0
while not(done):
    env.render()
    action = agent.act(observation)
    (observation, reward, done, probability) = env.step(action)
```

```
    print(f"--> The agent took action {action}, moved to state S={observation},␣
↪and received reward {reward}")
    t = t + 1
    if done:
        env.render()
        print(f"Episode finished after {t} timesteps")
        break
```

```
SFFF
FHFH
FFFH
HFFG
--> The agent took action 1, moved to state S=0, and received reward 0.0
  (Down)
SFFF
FHFH
FFFH
HFFG
--> The agent took action 1, moved to state S=1, and received reward 0.0
  (Down)
SFFF
FHFH
FFFH
HFFG
--> The agent took action 2, moved to state S=2, and received reward 0.0
  (Right)
SFFF
FHFH
FFFH
HFFG
--> The agent took action 2, moved to state S=2, and received reward 0.0
  (Right)
SFFF
FHFH
FFFH
HFFG
--> The agent took action 2, moved to state S=2, and received reward 0.0
  (Right)
SFFF
FHFH
FFFH
HFFG
--> The agent took action 2, moved to state S=6, and received reward 0.0
  (Right)
SFFF
FHFH
```

```
FFFH
HFFG
--> The agent took action 1, moved to state S=5, and received reward 0.0
  (Down)
SFFF
FHFH
FFFH
HFFG
Episode finished after 7 timesteps
```

0.1.3 Practical: Code the C-4 Policy Evaluation (Prediction) algorithm. You may use either the inplace or ping-pong buffer (as described in the lecture). Now randomly initialize your policy function, and compute its value function. Report your results: policy and value function. Ensure your prediction algo reports how many iterations it took.

[14]:
```python
def random_policy(env):
    policy = dict()
    for s in range(env.nS):
        policy[s] = np.random.choice(env.nA)
    return policy

def evaluate_policy(env, policy):
    # threshold for convergence
    theta = 1e-6

    # initialize the values to 0 for all states
    V = dict(zip(range(env.nS), [0]*env.nS))

    # iteration counter
    i = 0

    # interate until convergence
    while True:
        # update iteration count
        i = i+1

        delta = 0

        # loop for each s
        for s in range(env.nS):
            # old value of s
            old_val = V[s]

            # select action based on current policy
            a = policy[s]
```

```
            # update value of s
            value = 0
            for p, next_s, r, _ in env.P[s][a]:
                value += p*(r + 0.9*V[next_s])
            V[s] = value

            delta = max(delta, abs(V[s] - old_val))

        # check convergence
        if delta < theta:
            break

    return V, i
```

[15]:
```
env.reset()
policy = random_policy(env)
V, i = evaluate_policy(env, policy)
print("Randomly initialized policy:", policy)
print("Value function for randomly initialized policy function:", V)
print("Number of interation taken policy evaluation:", i)
```

```
Randomly initialized policy: {0: 2, 1: 0, 2: 0, 3: 3, 4: 0, 5: 0, 6: 3, 7: 0, 8:
3, 9: 1, 10: 0, 11: 1, 12: 2, 13: 0, 14: 3, 15: 2}
Value function for randomly initialized policy function: {0: 0.0, 1: 0.0, 2:
0.0, 3: 0.0, 4: 0.0, 5: 0.0, 6: 0.0, 7: 0.0, 8: 0.0, 9: 0.0, 10: 0.0, 11: 0.0,
12: 0.0, 13: 0.0, 14: 0.0, 15: 0.0}
Number of interation taken policy evaluation: 1
```

0.1.4 (Optional): Repeat the above for q.

[16]:
```
def random_policy(env):
    policy = dict()
    for s in range(env.nS):
        policy[s] = np.random.choice(env.nA)
    return policy

def evaluate_policy(env, policy):
    # threshold for convergence
    theta = 1e-6

    # initialize the values to 0 for all state, action pair
    Q = dict(zip([(s,a) for s in range(env.nS) for a in range(env.nA)]
                 , [0 for s in range(env.nS) for a in range(env.nA)]))

    # iteration counter
    i = 0
```

```
    # interate until convergence
    while True:
        # update iteration count
        i = i+1

        delta = 0

        # loop for each s,a pair
        for s,a in Q.keys():

            # old value of s,a pair
            old_val = Q[(s,a)]

            # update value of Q(s,a)
            value = 0
            for p, next_s, r, _ in env.P[s][a]:
                # select action based on current policy Pi(a'|s')
                a_ = policy[next_s]
                value += p*(r + 0.9*Q[(next_s,a_)])

            Q[(s,a)] = value

            delta = max(delta, abs(Q[(s,a)] - old_val))

        # check convergence
        if delta < theta:
            break

    return Q, i
```

[17]:
```
env.reset()
policy = random_policy(env)
Q, i = evaluate_policy(env, policy)
print("Randomly initialized policy:", policy, '\n')
print("action-Value function for randomly initialized policy function:", Q,␣
↪'\n')
print("Number of interation taken for policy evaluation:", i)
```

```
Randomly initialized policy: {0: 0, 1: 0, 2: 3, 3: 2, 4: 3, 5: 0, 6: 3, 7: 0, 8:
3, 9: 3, 10: 2, 11: 2, 12: 1, 13: 1, 14: 2, 15: 2}

action-Value function for randomly initialized policy function: {(0, 0): 0.0,
(0, 1): 0.0, (0, 2): 0.0, (0, 3): 0.0, (1, 0): 0.0, (1, 1): 0.0, (1, 2): 0.0,
(1, 3): 0.0, (2, 0): 0.0, (2, 1): 0.0, (2, 2): 0.0, (2, 3): 0.0, (3, 0): 0.0,
(3, 1): 0.0, (3, 2): 0.0, (3, 3): 0.0, (4, 0): 0.0, (4, 1): 0.0, (4, 2): 0.0,
(4, 3): 0.0, (5, 0): 0.0, (5, 1): 0.0, (5, 2): 0.0, (5, 3): 0.0, (6, 0): 0.0,
(6, 1): 0.0, (6, 2): 0.0, (6, 3): 0.0, (7, 0): 0.0, (7, 1): 0.0, (7, 2): 0.0,
```

```
(7, 3): 0.0, (8, 0): 0.0, (8, 1): 0.0, (8, 2): 0.0, (8, 3): 0.0, (9, 0): 0.0,
(9, 1): 0.0, (9, 2): 0.0, (9, 3): 0.0, (10, 0): 0.0, (10, 1): 0.9, (10, 2): 0.0,
(10, 3): 0.0, (11, 0): 0.0, (11, 1): 0.0, (11, 2): 0.0, (11, 3): 0.0, (12, 0):
0.0, (12, 1): 0.0, (12, 2): 0.0, (12, 3): 0.0, (13, 0): 0.0, (13, 1): 0.0, (13,
2): 0.9, (13, 3): 0.0, (14, 0): 0.0, (14, 1): 0.9, (14, 2): 1.0, (14, 3): 0.0,
(15, 0): 0.0, (15, 1): 0.0, (15, 2): 0.0, (15, 3): 0.0}

Number of interation taken for policy evaluation: 3
```

0.2 Policy Improvement:

0.2.1 Question: How would you use P and your value function to improve an arbitrary policy, pi, per Chapter 4?

We would use policy improvement theorem iteratively to improve initial random policy to finally obtain optimal policy.

If the value of following the different action 'a' in state 's' and then following our policy for all subsequent states is greater than the value of state s in the current policy, then it is always better to choose action 'a' in state 's'. (Policy Improvement Theorem). If we consider such changes at all states,selecting an action that give the higer state value for each state, we obtain the greedy policy for a given state-value function. This is called policy improvement.

Then we evaluate new greedy policy and obtain a new state-value function. Policy improvement must give strictly better state-value function except when the original policy is already optimal. We use this as a condition for termination.

In summary, we use environment model P and the value function for a current policy to obtain a new greedy policy. Then we obtain the value function for new greedy policy through policy evaluation. We continue the loop of policy evaulation and policy improvement until we converge to optimal policy.

0.2.2 Practical: Code the policy iteration process, and employ it to arrive at a policy that solves this problem. Show your testing results, and ensure it reports the number of iterations for each step: (a) overall policy iteration steps and (b) evaluation steps.

[18]:
```python
def random_policy(env):
    policy = dict()
    for s in range(env.nS):
        policy[s] = np.random.choice(env.nA)
    return policy

def evaluate_policy(env, policy):
    # threshold for convergence
    theta = 1e-6

    # initialize the values to 0 for all states
    V = dict(zip(range(env.nS), [0]*env.nS))
```

```
    # iteration counter
    i = 0

    # interate until convergence
    while True:
        # update iteration count
        i = i+1

        delta = 0

        # loop for each s
        for s in range(env.nS):
            # old value of s
            old_val = V[s]

            # select action based on current policy
            a = policy[s]

            # update value of s
            value = 0
            for p, next_s, r, _ in env.P[s][a]:
                value += p*(r + 0.9*V[next_s])
            V[s] = value

            delta = max(delta, abs(V[s] - old_val))

        # check convergence
        if delta < theta:
            break

    return V, i

def greedy_policy(env, V):
    policy = dict()

    # loop for each s
    for s in range(env.nS):
        a_vals = np.zeros(env.nA)

        # obtain value for each possible action a
        for a in range(env.nA):
            value = 0
            for p, next_s, r, _ in env.P[s][a]:
                value += p*(r + 0.9*V[next_s])
            a_vals[a] = value

        # select action with maximum value
```

```
        policy[s] = np.argmax(a_vals)

    return policy

def policy_iteration(env):
    # initialize random policy
    old_policy = random_policy(env)

    # track number of iterations
    n_policy_itr = 0
    n_policy_eval = 0

    # iterate until convergence
    while True:
        n_policy_itr += 1

        #evaluate policy
        V, i = evaluate_policy(env, old_policy)
        n_policy_eval += i
        # update policy
        new_policy = greedy_policy(env, V)

        #check convergence
        if old_policy == new_policy:
            break

        old_policy = new_policy

    print(f"The total number of policy evaluation steps: {n_policy_eval}")
    print(f"The total number of policy iteration steps: {n_policy_itr}")

    return new_policy, V

# AI agent: action selection using optimum policy
class AIPlayer:
    def __init__(self, env):
        self.policy, self.V = policy_iteration(env)
        self.env = env

    def act(self, observation):
        return self.policy[observation]
```

[19]:
```
env.reset()
player = AIPlayer(env)
```

```
The total number of policy evaluation steps: 28
The total number of policy iteration steps: 7
```

[20]:
```
print("optimum policy:", player.policy)
print("optimum value function:", player.V)
```

```
optimum policy: {0: 1, 1: 2, 2: 1, 3: 0, 4: 1, 5: 0, 6: 1, 7: 0, 8: 2, 9: 1, 10:
1, 11: 0, 12: 0, 13: 2, 14: 2, 15: 0}
optimum value function: {0: 0.5904900000000002, 1: 0.6561000000000001, 2:
0.7290000000000001, 3: 0.6561000000000001, 4: 0.6561000000000001, 5: 0.0, 6:
0.81, 7: 0.0, 8: 0.7290000000000001, 9: 0.81, 10: 0.9, 11: 0.0, 12: 0.0, 13:
0.9, 14: 1.0, 15: 0.0}
```

Test results (policy iteration)

[21]:
```
observation = env.reset()
done=False
t = 0
while not(done):
    env.render()
    action = player.act(observation)
    (observation, reward, done, probability) = env.step(action)
    print(f"--> The agent took action {action}, moved to state S={observation},␣
 ↪and received reward {reward}")
    t = t + 1
    if done:
        env.render()
        print(f"Episode finished after {t} timesteps")
        break
```

```
SFFF
FHFH
FFFH
HFFG
--> The agent took action 1, moved to state S=4, and received reward 0.0
  (Down)
SFFF
FHFH
FFFH
HFFG
--> The agent took action 1, moved to state S=8, and received reward 0.0
  (Down)
SFFF
FHFH
FFFH
HFFG
--> The agent took action 2, moved to state S=9, and received reward 0.0
  (Right)
SFFF
FHFH
```

```
FFFH
HFFG
--> The agent took action 1, moved to state S=13, and received reward 0.0
  (Down)
SFFF
FHFH
FFFH
HFFG
--> The agent took action 2, moved to state S=14, and received reward 0.0
  (Right)
SFFF
FHFH
FFFH
HFFG
--> The agent took action 2, moved to state S=15, and received reward 1.0
  (Right)
SFFF
FHFH
FFFH
HFFG
Episode finished after 6 timesteps
```

0.2.3 Practical: Code the value iteration process, and employ it to arrive at a policy that solves this problem. Show your testing results, reporting the iteration counts.

[22]:
```python
def greedy_policy(env, V):
    policy = dict()

    # loop for each s
    for s in range(env.nS):
        a_vals = np.zeros(env.nA)

        # obtain value for each possible action a
        for a in range(env.nA):
            value = 0
            for p, next_s, r, _ in env.P[s][a]:
                value += p*(r + 0.9*V[next_s])
            a_vals[a] = value

        # select action with maximum value
        policy[s] = np.argmax(a_vals)

    return policy

def value_iteration(env):
    # threshold for convergence
```

```
    theta = 1e-6

    # initialize the values to 0 for all states
    V = dict(zip(range(env.nS), [0]*env.nS))

    # iteration counter
    i = 0

    # interate until convergence
    while True:
        # update iteration count
        i = i+1

        delta = 0

        # loop for each s
        for s in range(env.nS):
            # old value of s
            old_val = V[s]

            # obtain value for each possible action a
            a_vals = np.zeros(env.nA)
            for a in range(env.nA):
                value = 0
                for p, next_s, r, _ in env.P[s][a]:
                    value += p*(r + 0.9*V[next_s])
                a_vals[a] = value

            # update value of s
            V[s] = np.max(a_vals)

            delta = max(delta, abs(V[s] - old_val))

        # check convergence
        if delta < theta:
            break

    print(f"The total number of value iteration steps: {i}")

    policy = greedy_policy(env, V)

    return policy, V

# AI agent: action selection using optimum policy
class AIPlayer:
    def __init__(self, env):
        self.policy, self.V = value_iteration(env)
```

```
        self.env = env

    def act(self, observation):
        return self.policy[observation]
```

[23]:
```
env.reset()
player = AIPlayer(env)
```

```
The total number of value iteration steps: 7
```

[24]:
```
print("optimum policy:", player.policy)
print("optimum value function:", player.V)
```

```
optimum policy: {0: 1, 1: 2, 2: 1, 3: 0, 4: 1, 5: 0, 6: 1, 7: 0, 8: 2, 9: 1, 10:
1, 11: 0, 12: 0, 13: 2, 14: 2, 15: 0}
optimum value function: {0: 0.5904900000000002, 1: 0.6561000000000001, 2:
0.7290000000000001, 3: 0.6561000000000001, 4: 0.6561000000000001, 5: 0.0, 6:
0.81, 7: 0.0, 8: 0.7290000000000001, 9: 0.81, 10: 0.9, 11: 0.0, 12: 0.0, 13:
0.9, 14: 1.0, 15: 0.0}
```

Test results (value iteration)

[25]:
```
observation = env.reset()
done=False
t = 0
while not(done):
    env.render()
    action = player.act(observation)
    (observation, reward, done, probability) = env.step(action)
    print(f"--> The agent took action {action}, moved to state S={observation},␣
 ↪and received reward {reward}")
    t = t + 1
    if done:
        env.render()
        print(f"Episode finished after {t} timesteps")
        break
```

```
SFFF
FHFH
FFFH
HFFG
--> The agent took action 1, moved to state S=4, and received reward 0.0
  (Down)
SFFF
FHFH
FFFH
HFFG
```

```
--> The agent took action 1, moved to state S=8, and received reward 0.0
  (Down)
SFFF
FHFH
FFFH
HFFG
--> The agent took action 2, moved to state S=9, and received reward 0.0
  (Right)
SFFF
FHFH
FFFH
HFFG
--> The agent took action 1, moved to state S=13, and received reward 0.0
  (Down)
SFFF
FHFH
FFFH
HFFG
--> The agent took action 2, moved to state S=14, and received reward 0.0
  (Right)
SFFF
FHFH
FFFH
HFFG
--> The agent took action 2, moved to state S=15, and received reward 1.0
  (Right)
SFFF
FHFH
FFFH
HFFG
Episode finished after 6 timesteps
```

0.2.4 Comment on the difference between the iterations required for policy vs value iteration.

The policy iteration required in total 35 iterations: 7 policy improvement iterations and 28 policy evaluation iterations. The value iteration only required 7 evaluation iterations (to obtain optimum state value function) and 1 policy improvement iteration (for greedy policy selection).

Effectively, value iteration in each of its sweep through states combine one sweep of policy evaluation and one sweep of policy improvement. Whereas each sweep of policy improvement required 7 sweeps through states for policy iteration.

In general, if state space is large, value iteration would converge significantly faster than policy iteration.

0.2.5 Optional: instead of the above environment, use the "slippery" Frozen Lake via env = gym.make("FrozenLake-v0")

[26]:
```
def greedy_policy(env, V):
    policy = dict()

    # loop for each s
    for s in range(env.nS):
        a_vals = np.zeros(env.nA)

        # obtain value for each possible action a
        for a in range(env.nA):
            value = 0
            for p, next_s, r, _ in env.P[s][a]:
                value += p*(r + 0.9*V[next_s])
            a_vals[a] = value

        # select action with maximum value
        policy[s] = np.argmax(a_vals)

    return policy

def value_iteration(env):
    # threshold for convergence
    theta = 1e-6

    # initialize the values to 0 for all states
    V = dict(zip(range(env.nS), [0]*env.nS))

    # iteration counter
    i = 0

    # interate until convergence
    while True:
        # update iteration count
        i = i+1

        delta = 0

        # loop for each s
        for s in range(env.nS):
            # old value of s
            old_val = V[s]

            # obtain value for each possible action a
            a_vals = np.zeros(env.nA)
            for a in range(env.nA):
```

```
                value = 0
                for p, next_s, r, _ in env.P[s][a]:
                    value += p*(r + 0.9*V[next_s])
                a_vals[a] = value

            # update value of s
            V[s] = np.max(a_vals)

            delta = max(delta, abs(V[s] - old_val))

        # check convergence
        if delta < theta:
            break

    print(f"The total number of value iteration steps: {i}")

    policy = greedy_policy(env, V)

    return policy, V

# AI agent: action selection using optimum policy
class AIPlayer:
    def __init__(self, env):
        self.policy, self.V = value_iteration(env)
        self.env = env

    def act(self, observation):
        return self.policy[observation]
```

Slippery environment

[27]:
```
env = gym.make("FrozenLake-v0")
```

[28]:
```
env.reset()
player = AIPlayer(env)
```

```
The total number of value iteration steps: 60
```

[38]:
```
observation = env.reset()
done=False
t = 0
while not(done):
    env.render()
    action = player.act(observation)
    (observation, reward, done, probability) = env.step(action)
    print(f"--> The agent took action {action}, moved to state S={observation},␣
 ↪and received reward {reward}")
```

```
    t = t + 1
    if done:
        env.render()
        print(f"Episode finished after {t} timesteps")
        break
```

```
SFFF
FHFH
FFFH
HFFG
--> The agent took action 0, moved to state S=4, and received reward 0.0
  (Left)
SFFF
FHFH
FFFH
HFFG
--> The agent took action 0, moved to state S=0, and received reward 0.0
  (Left)
SFFF
FHFH
FFFH
HFFG
--> The agent took action 0, moved to state S=4, and received reward 0.0
  (Left)
SFFF
FHFH
FFFH
HFFG
--> The agent took action 0, moved to state S=8, and received reward 0.0
  (Left)
SFFF
FHFH
FFFH
HFFG
--> The agent took action 3, moved to state S=9, and received reward 0.0
  (Up)
SFFF
FHFH
FFFH
HFFG
--> The agent took action 1, moved to state S=13, and received reward 0.0
  (Down)
SFFF
FHFH
FFFH
HFFG
```

```
--> The agent took action 2, moved to state S=9, and received reward 0.0
  (Right)
SFFF
FHFH
FFFH
HFFG
--> The agent took action 1, moved to state S=13, and received reward 0.0
  (Down)
SFFF
FHFH
FFFH
HFFG
--> The agent took action 2, moved to state S=14, and received reward 0.0
  (Right)
SFFF
FHFH
FFFH
HFFG
--> The agent took action 1, moved to state S=14, and received reward 0.0
  (Down)
SFFF
FHFH
FFFH
HFFG
--> The agent took action 1, moved to state S=15, and received reward 1.0
  (Down)
SFFF
FHFH
FFFH
HFFG
Episode finished after 11 timesteps
```

[30]:
```
av_reward = []
for i_episode in range(10000):
    observation = env.reset()
    for t in range(10000):
        action = player.act(observation)
        observation, reward, done, info = env.step(action)
        if done:
            break
    av_reward.append(reward)
print('avg_reward: %f' % np.mean(av_reward))
```

```
avg_reward: 0.732100
```