JUSTINE JOHN A. SERDONCILLO
IE 5371 DYNAMIC PROGRAMMING
& REINFORCEMENT LEARNING

OCTOBER 9, 2023
HOMEWORK #2

EXERCISE 3.7

{Robot running a maze} → +1 escape, episodic task; → 0 otherwise

Maximize $G_t \doteq R_{t+1} + R_{t+2} + \dots R_t$ (3.7) $\xRightarrow{\text{run}}$!! no improvement, why?

The reward is the same regardless of amount of time in the maze. Need to communicate that the faster to escape the better.

EXERCISE 3.8 $\gamma = 0.5$, $R_{n \in [1,5]} = [-1, 2, 6, 3, 2]$ $T = 5$ $G_{n \in [0,5]} = ?$

Using (3.8) discounted return $G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$

$G_5 = \sum_{k=0}^{\infty} (0.5)^k R_{6+k}$ but $R_{6+k} = 0$, $k = [0, \infty)$ then $G_5 = 0$

$G_4 = R_5 + \gamma G_5 = 2 + 0.5(0) = 2$

$G_3 = R_4 + \gamma(G_4) = 3 + 0.5(2) = 4$

$G_2 = R_3 + \gamma(G_3) = 6 + 0.5(4) = 8$

$G_1 = R_2 + \gamma(G_2) = 2 + 0.5(8) = 6$

$G_0 = R_1 + \gamma(G_1) = -1 + 0.5(6) = 2$

$\boxed{G_{n \in [0,5]} = [2, 6, 8, 4, 2, 0]}$

EXERCISE 3.9 $\gamma = 0.9$ $R_1 = 2$ $R_{[2,\infty)} = 7$ $G_0 \,\&\, G_1$?

using (3.8) $G_1 = \sum_{k=0}^{\infty} (0.9)^k R_{2+k} =$ infinite series of $7[1 + 0.9 + 0.9^2 + \dots] = 7 \cdot \frac{1}{1-0.9} = 70$

$G_0 = R_1 + \gamma G_1 = 2 + 0.9(70) = 65$

$\boxed{G_0, G_1 = 65, 70}$

Exercise 3.12 Give $v_\pi = f(q_\pi \,\&\, \pi)$

$$v_\pi = \underbrace{\sum_{a \in A(s)}}_{\text{sum through all actions}} \underbrace{\pi(a|s)}_{\text{probability of action } a \text{ given } s} \; q_\pi(s,a)$$

$\sim$ state-action value function

Exercise 3.13 Give $q_\pi = f(v_\pi, p)$

$$q_\pi = \underbrace{\sum_{s' \in S}}_{\text{sum all new states}} \underbrace{\sum_{r \in R}}_{\text{sum all rewards}} \underbrace{p(s', r | s, a)}_{\text{probability of new state \& reward given state \& action}} (\underbrace{r}_{\text{reward}} + \underbrace{\gamma}_{\text{discount}} \underbrace{v_\pi(s')}_{\text{value of new state}})$$

EXERCISE 3.15

Gridworld
- $r = $ "+" goals
- $r = $ "−" edge
- $r = 0$ otherwise

sgn(r) important? / intervals?

Prove w/ (3.8) $\forall r + = c \neq$ new values of States

find $V_c = V_c(c, \gamma)$

---

using (3.8)

$$G_t = \sum_{k=0}^{\infty} \gamma^k (R_{t+k+1})$$

now add c to all Rewards $\Rightarrow$ $G_t' = \sum_{k=0}^{\infty} \gamma^k (R_{t+k+1} + c)$

R_n can be "−", 0, "+"

can add c such that R_n is all "+", all same sign or "−"

$$G_t' = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} + \sum_{k=0}^{\infty} c\gamma^k = G_t + \frac{c}{1-\gamma}$$

$$\boxed{V_c = \frac{c}{1-\gamma}}$$

EXERCISE 3.17 Bellman equations for a, q_π

$$q_\pi(s,a) = f(q_\pi(s',a'))$$

Using result from EXERCISE 3.13 $q_\pi(s,a) = \sum_{s' \in S} \sum_{r \in R} p(s',r|s,a)\left[r + \gamma v_\pi(s')\right]$ (A)

& EXERCISE 3.12 $v_\pi(s) = \sum_{a \in A} \pi(a|s) \cdot q_\pi(s,a)$ (B)

plug in (B) to (A)

$$\boxed{q_\pi(s,a) = \sum_{s',r} p(s',r|s,a)\left[r + \sum_{a' \in A} \pi(a'|s') \cdot \gamma q_\pi(s',a')\right]}$$

EXERCISE 3.25. Give $v_* = v_*(q_*)$

$$v_*(s) = \max_{a \in A(s)} \underbrace{q_*(s,a)}_{\text{optimal q-factor}}$$

EXERCISE 3.26 Give $q_* = q_*(v_*, p)$.

$$q_*(s,a) = \max \underbrace{\sum_{s' \in S} \sum_{r \in R} p(s', r \mid s,a) \left[r + \gamma v_*(s') \right]}_{\text{best value}}$$

EXERCISE 3.27 Give $\pi_* = \pi_*(q_*)$

$$\pi_*(a \mid s) = \max_{a \in A} \underbrace{q_*(s',a)}_{\substack{\text{best state-action} \\ \text{value}}}$$

EXERCISE 4.7 Write a program for policy iteration & resolve Jack's car rental problem w/ the following changes

① Jack's employee at 1st rides bus & lives in 2nd. Shuttle 1 car to 2nd for free

② 10 cars limit at each location, 4$ cost for extra

From Example 4.2 Jack's Car Rental

	①		②	
requests	$P(n) = \frac{3^n}{n!} e^{-3}$	Poisson	$P(n) = \frac{4^n}{4!} e^{-4}$	10$ per car rented
return	$p(n) = \frac{3^n}{n!} e^{-3}$		$P(n) = \frac{2^n}{2!} e^{-2}$	2$ to move / car
	max 20		max 20	max 5 cars move / night
				$\gamma = 0.9$

Also use Policy Iteration Method.

EXERCISE 4.5 How would policy iteration be defined for action values? Give a complete algorithm for computing q_*, analogous to that on page 80 for computing v_*. Please pay special attention to this exercise, because the ideas involved will be used throughout the rest of the book.

Policy iteration for action values is essentially trying to find the best policy

1. Initialization

$V(s) \in \mathbb{R}$ & $\pi(s) = A(s)$ arbitrarily $\forall s \in S$; $V(\text{terminal}) = 0$

$Q(s,a)$ arbitrarily $\forall s \in S$, $a \in A(s)$

set learning rate & discount

while Q not converged

while S not terminal

$\pi(s) = \underset{a \in A(s)}{\operatorname{argmax}} Q(s,a)$

Store action, reward & new state

Calculate new $Q(s',a)$ based on $Q(s,a)$ & learning rate

Set s as the s'

return Q

Justine John "JJ" A. Serdoncillo **EE 5571: Reinforcement Learning and Dynamic Programming**
October 9, 2023 **Monday**

Jack's Capitalism Optimization

Exercise 4.7: Jack's normal day

It can be seen from the figures below that these photos match with the photos found from the book. Both of them stopped after iteration 4 and the corresponding value function using the optimized policy can be seen as well!

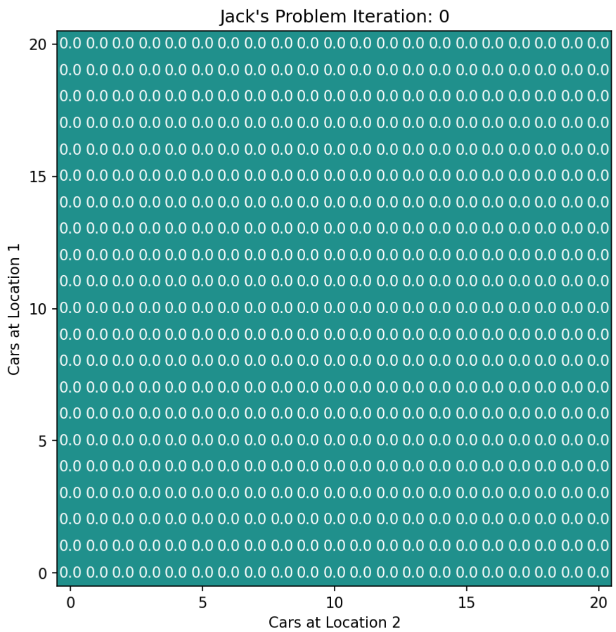

Figure 1: Initialized Policy

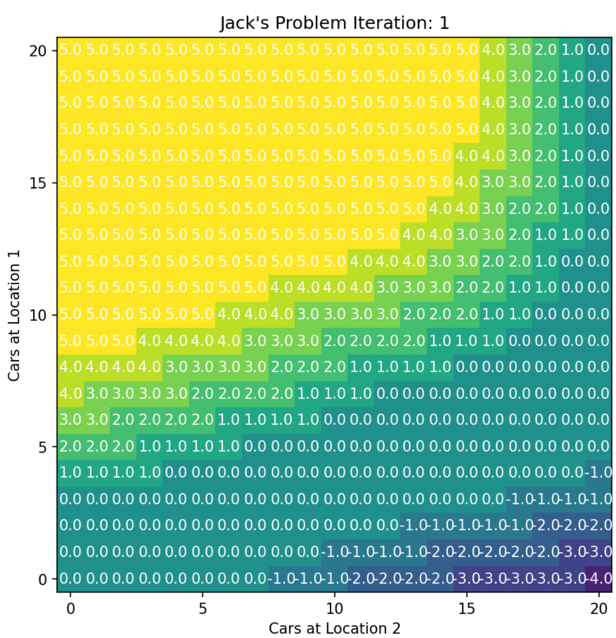

Figure 2: Policy after 1 iteration

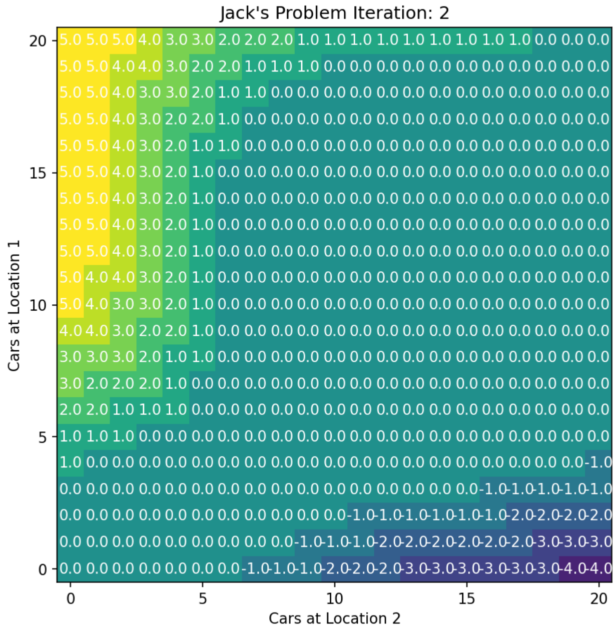

Figure 3: Policy after 2 iteration

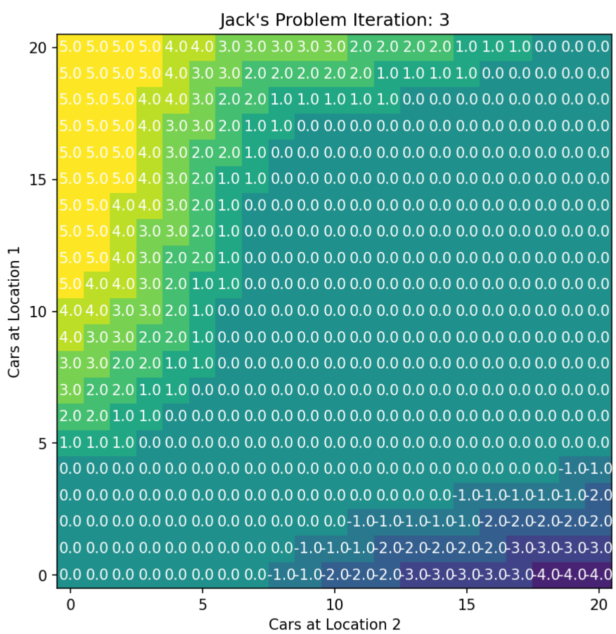

Figure 4: Policy after 3 iteration

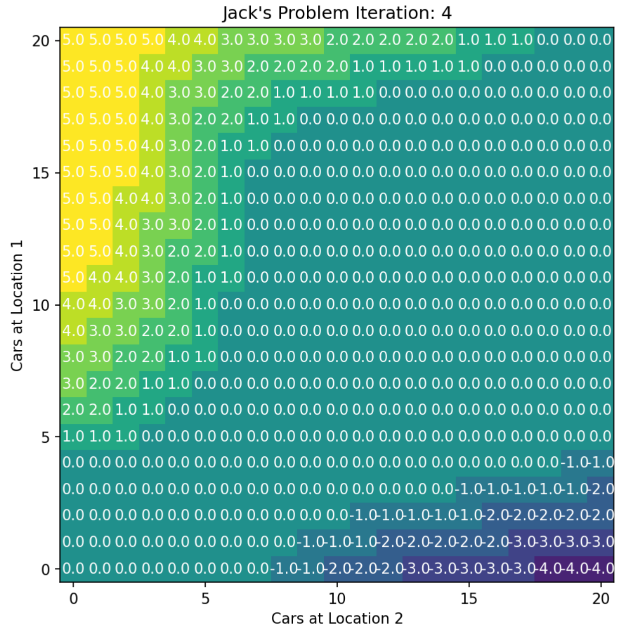

Figure 5: Policy after 4 iteration

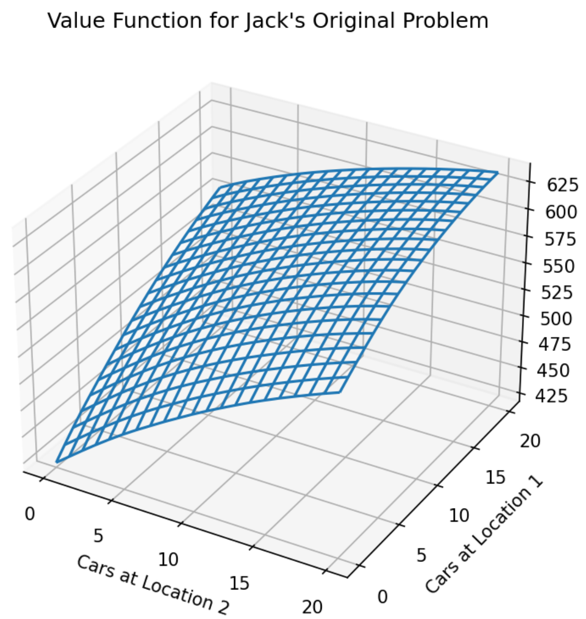

Figure 6: Value with optimized policy

Exercise 4.7: Jack's special day

For the special problem, there was a couple of extra steps that needed to be coded but overall the process is still the same. Starting with Policy Evaluation and then Policy Iteration in order to get a "steady-state" policy. It can be seen that the final policy looks different from the original problem and that the full on 20 doesn't have a full on move 5 anymore. This can be due to the fact that it tries to avoid the extra payment or it can also take advantage of it because of the fixed rate.

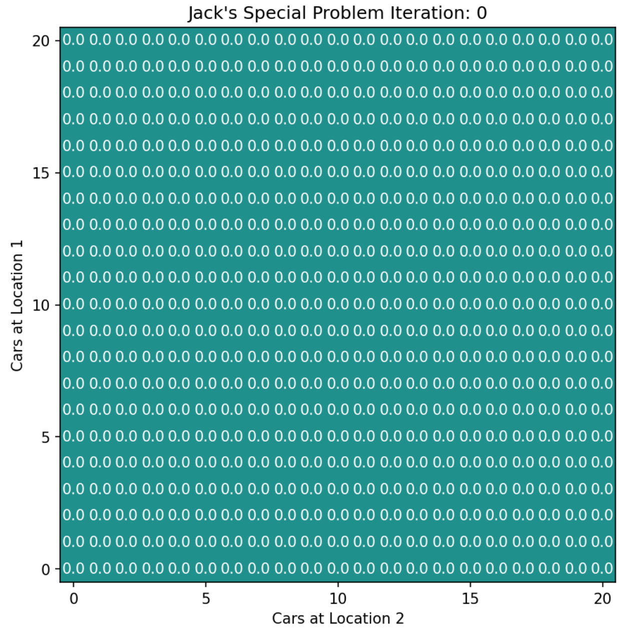

Figure 7: Initialized Policy

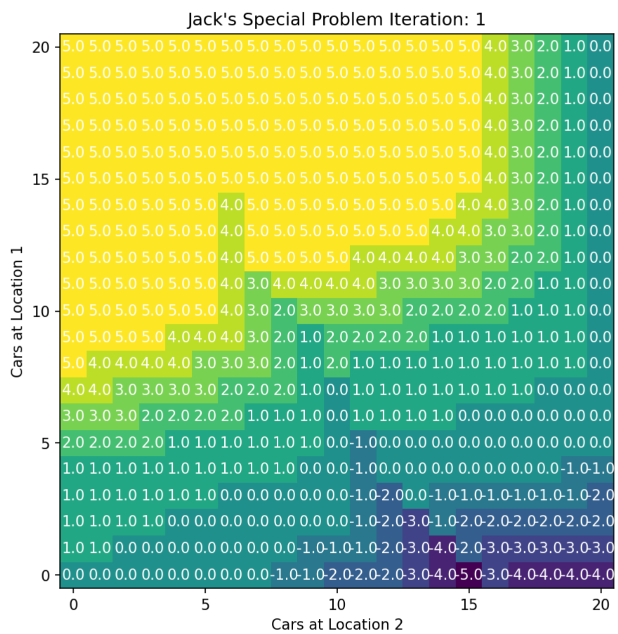

Figure 8: Policy after 1 iteration

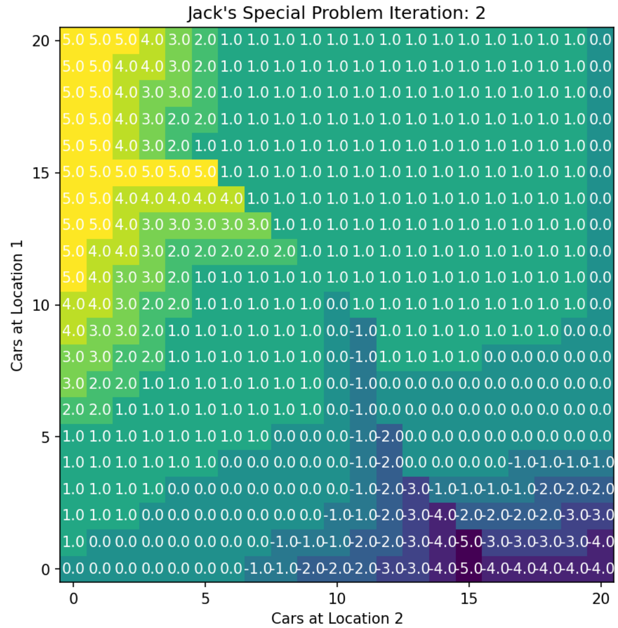

Figure 9: Policy after 2 iteration

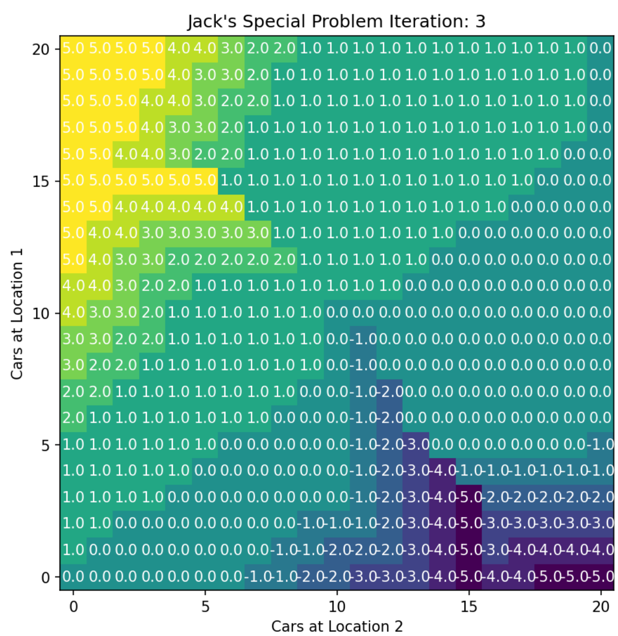

Figure 10: Policy after 3 iteration

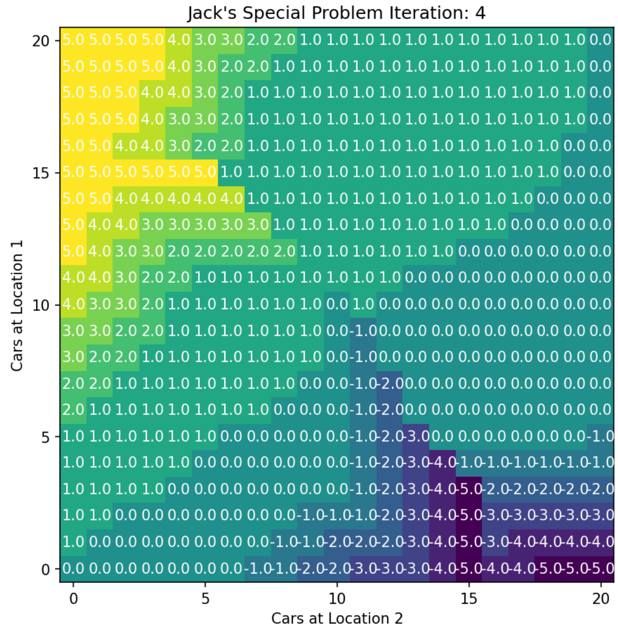

Figure 11: Policy after 4 iteration

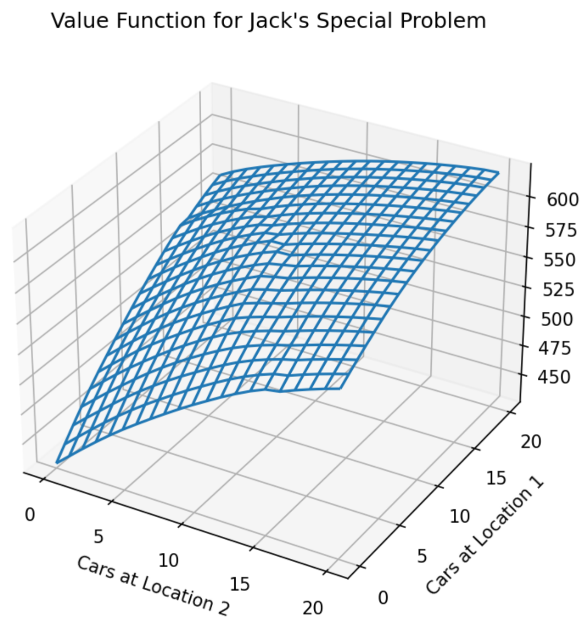

Figure 12: Value with optimized policy

Appendix

Python Code for Original

```
'''
Justine Serdoncillo
IE 5571 - Dynamic Programming
HW 2 Exercise 4.7
October 9, 2023
'''

"""
Write a program for policy iteration and re-solve J a c k s car
rental problem with the following changes. One of J a c k s employees at the first location
rides a bus home each night and lives near the second location. She is happy to shuttle
one car to the second location for free. Each additional car still costs $2, as do all cars
moved in the other direction. In addition, Jack has limited parking space at each location.
If more than 10 cars are kept overnight at a location (after any moving of cars), then an
additional cost of $4 must be incurred to use a second parking lot (independent of how
many cars are kept there). These sorts of nonlinearities and arbitrary dynamics often
occur in real problems and cannot easily be handled by optimization methods other than
dynamic programming. To check your program, first replicate the results given for the
original problem.
"""

import numpy as np
import matplotlib.pyplot as plt
import random
import math

# %% Use the Poisson Equation for the probability
def poisson(lam, n):
    return lam**n * math.exp(-lam) / math.factorial(n)

# Return a list of all possible actions
def possibleActions(state):
    # Get the state of the cars
    firLoc = state[0]
    secLoc = state[1]
    #breakpoint()
    # list all possible & impossible actions
    allActions = [x for x in range(-maxMove, maxMove+1)]
    possibleActions = []

    for action in allActions:
        newFir = firLoc - action
        newSec = secLoc + action
        if  newFir > maxCar or newFir < 0 or newSec > maxCar or newSec < 0:
            pass
        else:
            possibleActions.append(action)

    return possibleActions

# %% Evaluate a state and action based on a policy
def evalVal(state, action):
    # Initialize
    value = 0

    # Get the new state of the cars and compute the cost
    firLoc = state[0] - action
    secLoc = state[1] + action
    cost = costCar * abs(action)

    # Summation over all the possible new states
    ps1req, ps2req, ps1ret, ps2ret = 1, 1, 1, 1
    for reqs1 in range(firLoc+1):
        p1 = poisson(3, reqs1) if reqs1 != firLoc else ps1req
        ps1req -= p1
        g = maxCar + reqs1 - firLoc
        r1 = creditCar * p1
```

```
        for reqs2 in range(secLoc+1):
            p2 = poisson(4, reqs2) if reqs2 != secLoc else ps2req
            ps2req -= p2
            h = maxCar + reqs2 - secLoc
            r2 = creditCar * p2
            for rets1 in range(g + 1):
                p3 = poisson(3, rets1) if rets1 != g else ps1ret
                ps1ret -= p3
                for rets2 in range(h + 1):
                    p4 = poisson(2, rets2) if rets2 != h else ps2ret
                    ps2ret -= p4

                    # Sum all over the possible rewards
                    pTot = p1 * p2 * p3 * p4
                    reward = r1 + r2 - cost + gamma*values[firLoc-reqs1+rets1, secLoc-reqs2+rets2]
                    value += pTot * reward
    #breakpoint()
    return value

# %% Evaluate the Value of the policy
def policyEvaluation():
    print("Policy Evaluation")
    print("=================")
    ite = 0
    maxIte = 10
    while ite < maxIte:
        ite += 1
        delta = 0
        #breakpoint()
        for s in states:
            v = values[s]
            values[s] = evalVal(s, policy[s])
            #breakpoint()
            delta = max(delta, abs(v - values[s]))
        print(f"Current Delta at Iteration {ite}: {delta}")
        if delta < theta:
            break

# %% Actual Problem statement
# Problem parameters
maxCar = 20
maxMove = 5
creditCar = 10
costCar = 2

# Learning Parameters
gamma = 0.9
theta = 1E-2
maxIte = 5
ite = 0

# list comprehension for the states
states = [(x,y) for x in range(maxCar+1) for y in range(maxCar+1)]
policy = np.zeros((maxCar+1, maxCar+1), dtype=np.int8)
values = np.zeros((maxCar+1, maxCar+1))
#policy[10,0] = 5

fig, ax = plt.subplots(figsize=(8,6), dpi=150)
ax.imshow(policy, origin="lower", interpolation='none', vmin=-maxMove, vmax=maxMove)
ax.set_title(f"Jack's Problem Iteration: {ite}")
fig.tight_layout()
ax.set_xlabel("Cars at Location 2")
ax.set_ylabel("Cars at Location 1")
ax.set_xticks(np.arange(0, maxCar+1, 5))
ax.set_yticks(np.arange(0, maxCar+1, 5))
for yy in range(policy.shape[1]):
    for xx in range(policy.shape[0]):
        text = ax.text(xx, yy, policy[yy, xx], ha="center", va="center", color="w")

while ite < maxIte:
    ite += 1
```

```
    print(f"Iteration: {ite}")
    print( "~~~~~~~~~~~~~~~~")

    # Policy Evaluation
    print("Policy Evaluation")
    print("=================")
    ite = 0
    maxIte = 50
    while ite < maxIte:
        ite += 1
        delta = 0
        #breakpoint()
        for s in states:
            v = values[s]
            values[s] = evalVal(s, policy[s])
            #breakpoint()
            delta = max(delta, abs(v - values[s]))
        print(f"Current Delta at Iteration {ite}: {delta}")
        if delta < theta:
            break

    # Policy Improvement
    policy_stable = True
    for s in states:
        # copy policy to become old policy
        old = policy[s].copy()

        # create values dictionary based on chosen action
        vvalues = {a: evalVal(s,a) for a in possibleActions(s)}
        # random choose an action from the actions that map to the max values
        bestActions = [a for a,value in vvalues.items() if value == np.max(list(vvalues.values()))]
        policy[s] = np.random.choice(bestActions)

        # compare if old action is same as current action
        if old != policy[s]:
            policy_stable = False
    if policy_stable:
        break

    # Visualize current policy
    fig, ax = plt.subplots(figsize=(8,6), dpi=150)
    ax.imshow(policy, origin="lower", interpolation='none', vmin=-maxMove, vmax=maxMove)
    ax.set_title(f"Jack's Problem Iteration: {ite}")
    fig.tight_layout()
    ax.set_xlabel("Cars at Location 1")
    ax.set_ylabel("Cars at Location 2")
    ax.set_xticks(np.arange(0, maxCar+1, 5))
    ax.set_yticks(np.arange(0, maxCar+1, 5))
    for yy in range(policy.shape[1]):
        for xx in range(policy.shape[0]):
            text = ax.text(xx, yy, policy[yy, xx], ha="center", va="center", color="w")


# Print the optimal value function
fig = plt.figure(figsize=(8,6), dpi=150)
ax = plt.axes(projection='3d')
X, Y = np.meshgrid(range(maxCar+1), range(maxCar+1))
ax.plot_wireframe(X, Y, values, rstride=1, cstride=1)
ax.set_xlabel("Cars at Location 2")
ax.set_ylabel("Cars at Location 1")
ax.set_zlabel("V", rotation=0)
ax.set_title("Value Function for Jack's Original Problem")
ax.set_xticks(np.arange(0, maxCar+1, 5))
ax.set_yticks(np.arange(0, maxCar+1, 5))
fig.savefig("values.png")
```

Python Code for Special

```
'''
Justine Serdoncillo
IE 5571 - Dynamic Programming
HW 2 Exercise 4.7
October 9, 2023
'''

"""
Write a program for policy iteration and re-solve  J a c k s  car
rental problem with the following changes. One of  J a c k s  employees at the first location
rides a bus home each night and lives near the second location. She is happy to shuttle
one car to the second location for free. Each additional car still costs $2, as do all cars
moved in the other direction. In addition, Jack has limited parking space at each location.
If more than 10 cars are kept overnight at a location (after any moving of cars), then an
additional cost of $4 must be incurred to use a second parking lot (independent of how
many cars are kept there). These sorts of nonlinearities and arbitrary dynamics often
occur in real problems and cannot easily be handled by optimization methods other than
dynamic programming. To check your program, first replicate the results given for the
original problem.
"""

import numpy as np
import matplotlib.pyplot as plt
import random
import math

# %% Use the Poisson Equation for the probability
def poisson(lam, n):
    return lam**n * math.exp(-lam) / math.factorial(n)

# Return a list of all possible actions
def possibleActions(state):
    # Get the state of the cars
    firLoc = state[0]
    secLoc = state[1]
    #breakpoint()
    # list all possible & impossible actions
    allActions = [x for x in range(-maxMove, maxMove+1)]
    possibleActions = []

    for action in allActions:
        newFir = firLoc - action
        newSec = secLoc + action
        if  newFir > maxCar or newFir < 0 or newSec > maxCar or newSec < 0:
            pass
        else:
            possibleActions.append(action)

    return possibleActions

# %% Evaluate a state and action based on a policy
def evalVal(state, action):
    # Initialize
    value = 0

    # Get the new state of the cars and compute the cost
    firLoc = state[0] - action
    secLoc = state[1] + action
    if action > 0:
        action -= 1
        cost = costCar * abs(action)
    else:
        if firLoc > limCar or secLoc > limCar:
            sad = 1
        else:
            sad = 0
        cost = costCar * abs(action) + overCost * sad

    # Summation over all the possible new states
    ps1req, ps2req, ps1ret, ps2ret = 1, 1, 1, 1
```

```
    for reqs1 in range(firLoc+1):
        p1 = poisson(3, reqs1) if reqs1 != firLoc else ps1req
        ps1req -= p1
        g = maxCar + reqs1 - firLoc
        r1 = creditCar * p1
        for reqs2 in range(secLoc+1):
            p2 = poisson(4, reqs2) if reqs2 != secLoc else ps2req
            ps2req -= p2
            h = maxCar + reqs2 - secLoc
            r2 = creditCar * p2
            for rets1 in range(g + 1):
                p3 = poisson(3, rets1) if rets1 != g else ps1ret
                ps1ret -= p3
                for rets2 in range(h + 1):
                    p4 = poisson(2, rets2) if rets2 != h else ps2ret
                    ps2ret -= p4

                    # Sum all over the possible rewards
                    pTot = p1 * p2 * p3 * p4
                    reward = r1 + r2 - cost + gamma*values[firLoc-reqs1+rets1, secLoc-reqs2+rets2]
                    value += pTot * reward
    return value

# %% Evaluate the Value of the policy
def policyEvaluation():
    print("Policy Evaluation")
    print("=================")
    ite = 0
    maxIte = 10
    while ite < maxIte:
        ite += 1
        delta = 0
        #breakpoint()
        for s in states:
            v = values[s]
            values[s] = evalVal(s, policy[s])
            #breakpoint()
            delta = max(delta, abs(v - values[s]))
        print(f"Current Delta at Iteration {ite}: {delta}")
        if delta < theta:
            break

# %% Actual Problem statement
# Problem parameters
maxCar = 20
maxMove = 5
creditCar = 10
costCar = 2

# Extra Rules
limCar = 10
overCost = 4

# Learning Parameters
gamma = 0.9
theta = 1E-2
maxIte = 0
ite = 0
stay = True

# list comprehension for the states
states = [(x,y) for x in range(maxCar+1) for y in range(maxCar+1)]
policy = np.zeros((maxCar+1, maxCar+1), dtype=np.int8)
values = np.zeros((maxCar+1, maxCar+1))
#policy[10,0] = 5

fig, ax = plt.subplots(figsize=(8,6), dpi=150)
ax.imshow(policy, origin="lower", interpolation='none', vmin=-maxMove, vmax=maxMove)
ax.set_title(f"Jack's Problem Iteration: {ite}")
fig.tight_layout()
ax.set_xlabel("Cars at Location 2")
```

```
ax.set_ylabel("Cars at Location 1")
ax.set_xticks(np.arange(0, maxCar+1, 5))
ax.set_yticks(np.arange(0, maxCar+1, 5))
for yy in range(policy.shape[1]):
    for xx in range(policy.shape[0]):
        text = ax.text(xx, yy, policy[yy, xx], ha="center", va="center", color="w")

while ite < maxIte:
    ite += 1
    print(f"Iteration: {ite}")
    print( "~~~~~~~~~~~~~~~~")

    # Policy Evaluation
    print("Policy Evaluation")
    print("=================")
    ite = 0
    maxIte = 50
    while ite < maxIte:
        ite += 1
        delta = 0
        #breakpoint()
        for s in states:
            v = values[s]
            values[s] = evalVal(s, policy[s])
            #breakpoint()
            delta = max(delta, abs(v - values[s]))
        print(f"Current Delta at Iteration {ite}: {delta}")
        if delta < theta:
            break

    # Policy Improvement
    policy_stable = True
    for s in states:
        # copy policy to become old policy
        old = policy[s].copy()

        # create values dictionary based on chosen action
        vvalues = {a: evalVal(s,a) for a in possibleActions(s)}
        # random choose an action from the actions that map to the max values
        bestActions = [a for a,value in vvalues.items() if value == np.max(list(vvalues.
    values()))]
        policy[s] = np.random.choice(bestActions)

        # compare if old action is same as current action
        if old != policy[s]:
            policy_stable = False
    if policy_stable:
        break

    # Visualize current policy
    fig, ax = plt.subplots(figsize=(8,6), dpi=150)
    ax.imshow(policy, origin="lower", interpolation='none', vmin=-maxMove, vmax=maxMove)
    ax.set_title(f"Jack's Special Problem Iteration: {ite}")
    fig.tight_layout()
    ax.set_xlabel("Cars at Location 1")
    ax.set_ylabel("Cars at Location 2")
    ax.set_xticks(np.arange(0, maxCar+1, 5))
    ax.set_yticks(np.arange(0, maxCar+1, 5))
    for yy in range(policy.shape[1]):
        for xx in range(policy.shape[0]):
            text = ax.text(xx, yy, policy[yy, xx], ha="center", va="center", color="w")


# Print the optimal value function
fig = plt.figure(figsize=(8,6), dpi=150)
ax = plt.axes(projection='3d')
X, Y = np.meshgrid(range(maxCar+1), range(maxCar+1))
ax.plot_wireframe(X, Y, values, rstride=1, cstride=1)
ax.set_xlabel("Cars at Location 2")
ax.set_ylabel("Cars at Location 1")
ax.set_zlabel("V", rotation=0)
ax.set_title("Value Function for Jack's Original Problem")
```

```
ax.set_xticks(np.arange(0, maxCar+1, 5))
ax.set_yticks(np.arange(0, maxCar+1, 5))
fig.savefig("values.png")
```