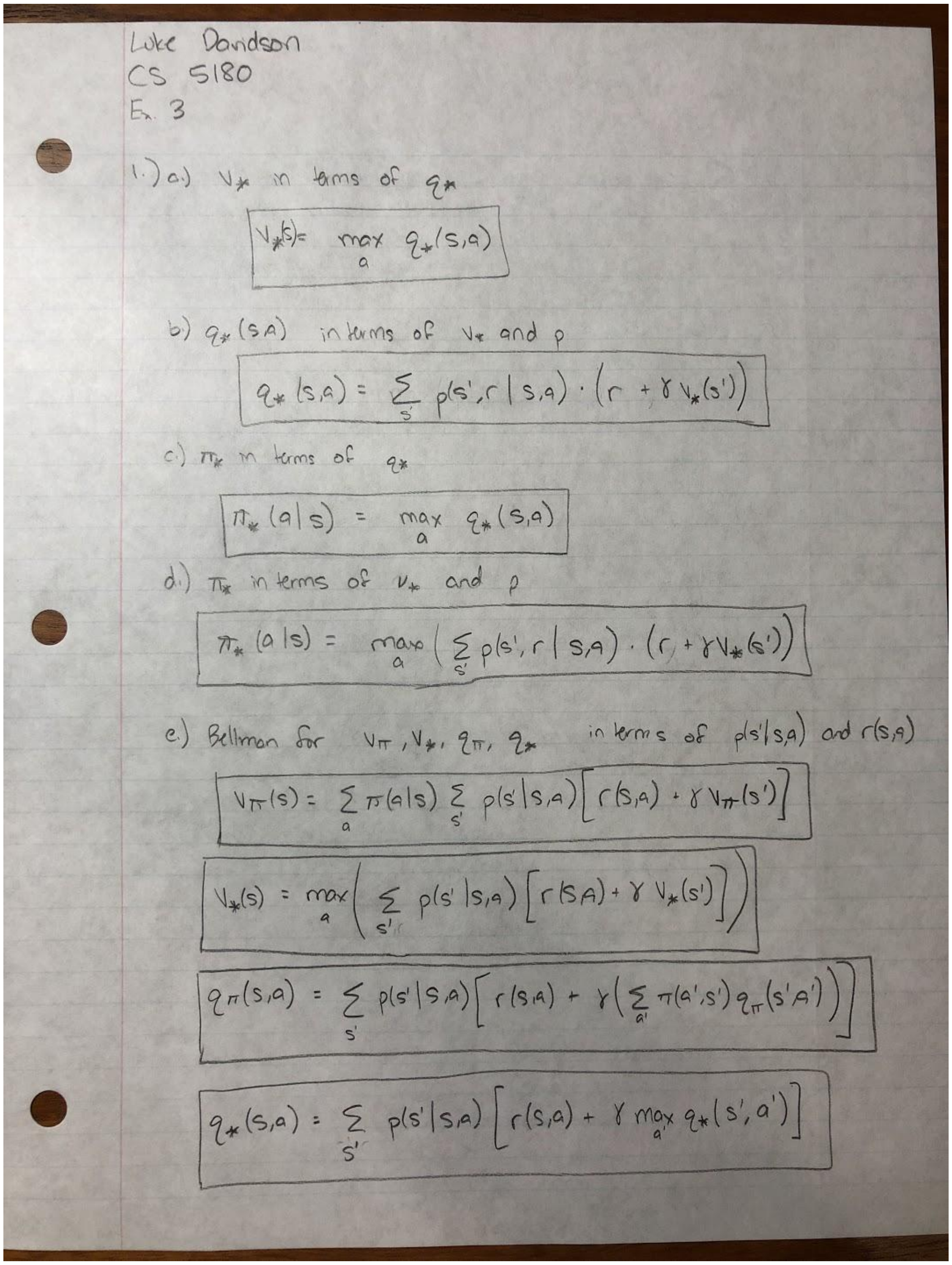
Luke Davidson
CS 5180
Ex3



Luke Davidson
CS 5180
Ex. 3

1.) a.) v_* in terms of q_*

$$\boxed{v_*(s) = \max_a q_*(s,a)}$$

b.) $q_*(s,A)$ in terms of v_* and p

$$\boxed{q_*(s,a) = \sum_{s'} p(s',r \mid s,a) \cdot \left(r + \gamma v_*(s')\right)}$$

c.) π_* in terms of q_*

$$\boxed{\pi_*(a \mid s) = \max_a q_*(s,a)}$$

d.) π_* in terms of v_* and p

$$\boxed{\pi_*(a \mid s) = \max_a \left(\sum_{s'} p(s',r \mid s,a) \cdot \left(r + \gamma v_*(s')\right)\right)}$$

e.) Bellman for v_π, v_*, q_π, q_* in terms of $p(s'|s,a)$ and $r(s,a)$

$$\boxed{v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} p(s'|s,a)\left[r(s,a) + \gamma v_\pi(s')\right]}$$

$$\boxed{v_*(s) = \max_a \left(\sum_{s'} p(s'|s,a)\left[r(s,A) + \gamma v_*(s')\right]\right)}$$

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

$$\boxed{q_*(s,a) = \sum_{s'} p(s'|s,a)\left[r(s,a) + \gamma \max_{a'} q_*(s',a')\right]}$$

2.) a) For action values, policy iteration would be defined by Q_π instead of V_π. So the process would look similar to

$$\pi_0 \xrightarrow{E} Q_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} Q_{\pi_1} \xrightarrow{I} \pi_2 \longrightarrow Q_{\pi_2} \longrightarrow \ldots$$

E = evaluate
I = improve

algorithm would look similar to that in the textbook / notes for V_π.

```
1.) Initialize

    $\pi(s) \in A(s)$  for s in {S}
    $Q(s,a)$
    difference = 0
2.) Evaluate

    while convergence not reached: (difference > 0.001)
        for each state in {S}
            for each action in A(state):
                q_old = Q(state, action)
                $Q(state, action) = \sum_{s'} p(s'|s,a) [ r(s'|s,a) + \gamma Q(s', \pi(s)) ]$
                difference = max( difference, | q_old - Q(s,a) )

3.) Improvement

    for state in {S}:
        action = $\pi$(state)
        $\pi$(state) = $\max_a Q$(state, action)
        if action != $\pi$(state)
            repeat evaluate step
    else
        return $\pi(s)$. as $\pi_*(s)$.
```

2b.) eq 4.10 = $v_{k+1}(s) = \max_a \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_k(s')\right]$

for $q_{k+1}(s)$ it'll be similar but no max over a, it will simply be a sum over s' given s, a

$$\boxed{q_{k+1}(s) = \sum_{s'} p(s',r|s,a)\left[r + \gamma\, q_k(s',a')\right]}$$

3.) $\{s\} = \{x, y, z\}$
$\hookrightarrow$ terminal

s	a	s'	p	r(s')
x	b	y	0.8	-2
x	b	x	0.2	-1
y	b	x	0.8	-1
y	b	y	0.2	-2
x	c	z	0.1	0
y	c	z	0.1	0
x	c	x	0.9	-1
y	c	y	0.9	-2

a.) It's evident that the main goal of the agent is to reach the terminal state "z" as fast as possible since all other actions / states will produce a negative reward. However, in choosing action "c", there is a very high chance the agent still receives a negative reward. In the long run, the agent will want to maximize the sum of rewards it receives, or in this case, minimize how negative the cumulative sum gets.

To do this, the agent will likely favor going to state x as the reward is less negative than state y and therefore it is a better option to try to reach state z from. In other words, action b will likely be favored in state y as opposed to action c, even though action c will get to the terminal goal faster. "b" will be favored because there is a high chance of going to state x (0.8), which is only a -1 reward. Action c will be more likely in state x than state y because of a similar reason (-1 reward 90% likely in x, compared to -2 90% likely in y).

3b.) ① Initialize

$V(s) = [0, 0, 0]$
$\quad\quad\quad x \;\; y \;\; z$

$\pi = [c, c]$

② Evaluate

$$v(x|c) = \underbrace{(0.1)(0 + \gamma(0))}_{state' = z} + \underbrace{(0.9)(-1 + \gamma(0))}_{state' = x} = \boxed{-0.9}$$

$$v(y|c) = (0.1)(0 + \gamma(0)) + (0.9)(-2 + \gamma(0)) = \boxed{-1.8}$$

$$v(z|c) = \boxed{0}$$

③ update

$$\pi(x) = \max\{x|b, x|c\} = \max\big(\underbrace{(0.8)(-2 + \gamma(-1.8))}_{x|b \rightarrow \; 0.8 \rightarrow y,\; 0.2 \rightarrow x} + (0.2)(-1 -$$

$$x|b = (0.8)(-3.8) + (0.2)(-1.9) = -3.42$$

$$x|c = (0.9)(-1 - 0.9) + (0.1)(0) = -1.71$$

$$\boxed{\pi(x) = c}$$

$$\pi(y) = \max\{y|b, y|c\} = \begin{cases} y|b = (0.8)(-1 - 0.9) + (0.2)(-2 - 1.8) = -2.28 \\ y|c = (0.9)(-2 - 1.8) + (0.1)(0) = -3.42 \end{cases}$$

$$\boxed{\pi(y) = b}$$

Since the old policy for state y changed from c → b, we go again, now with

$V(s) = [x, y, z] = [-0.9, -1.8, 0]$
$\pi = [x, y] = [c, b]$

① Evaluate

$V(x|c) = (0.9)(-1-0.9) + (0.1)(0+0) = -1.71$
$V(y|b) = (0.8)(-1-0.9) + (0.2)(-2-1.8) = -2.28$

② Improve w/ $V(s) = [-1.71, -2.28, 0]$

$$\pi(x) = \max \begin{Bmatrix} x|b \\ x|c \end{Bmatrix} = \begin{cases} (0.8)(-2-2.28) + (0.2)(-1-1.71) = \boxed{-3.966} \\ (0.9)(-1-1.71) = \boxed{-2.439} \end{cases}$$
$$\pi(y) = \max \begin{Bmatrix} y|b \\ y|c \end{Bmatrix} = \begin{cases} (0.8)(-1-1.71) + (0.2)(-2-2.28) = \boxed{-3.024} \\ (0.9)(-2-2.28) = \boxed{-3.852} \end{cases}$$

so $\pi(x) = c$
$\pi(y) = b$ $\longrightarrow$ there is no change, so report

$V_* = \{ "x" = -1.71, "y" : -2.28, "z" : 0 \}$
$\pi_* = \{ "x" = c, "y" = b \}$

3c.) if $\pi_0(s) = [\overset{x}{b}, \overset{y}{b}]$

$v(s) = [0, 0, 0]$

<u>eval</u>

$v(x|b) = (0.8)(-2) + (0.2)(-1) = -1.8$
$v(y|b) = (0.8)(-1) + (0.2)(-2) = -1.2$

<u>improve</u>

$$\pi(x) = \max \begin{cases} x|b : (0.8)(-2-1.2) + (0.2)(-1-1.8) = -3.12 \\ x|c : (0.9)(-1-1.8) = \boxed{-2.52} \end{cases}$$

$$\pi(y) = \max \begin{cases} y|b = (0.8)(-1-1.8) + (0.2)(-2,-1.2) = \boxed{-2.88} \\ y|c = (0.9)(-2-1.2) = \boxed{-2.88} \end{cases}$$

$\pi(x)$ still says action c, but $\pi(y)$ clearly shows a tie, resulting in a random choice between b and c. This will repeat, causing a the π^* to not be reached. Discounting will help eliminate this error because it will slightly change the values of each state by the discount factor. However, yes, the optimal policy will depend on what the value of the discount factor is. The discount factor essentially tells the agent to care more about the long term or short term. Depending on the value, the agent will want to optimally act in a different way, leading to different optimal policies.

4.) a.) Results:

```
==============================
==  Optimal State Value    ==
==============================
[[22.  24.4 22.  19.4 17.5]
 [19.8 22.  19.8 17.8 16. ]
 [17.8 19.8 17.8 16.  14.4]
 [16.  17.8 16.  14.4 13. ]
 [14.4 16.  14.4 13.  11.7]]
==============================
==============================
==     Optimal Policy      ==
==============================
[0, 0] = ['east']
[0, 1] = ['north', 'south', 'west', 'east']
[0, 2] = ['west']
[0, 3] = ['north', 'south', 'west', 'east']
[0, 4] = ['west']
------------------------------
[1, 0] = ['north', 'east']
[1, 1] = ['north']
[1, 2] = ['north', 'west']
[1, 3] = ['west']
[1, 4] = ['west']
------------------------------
[2, 0] = ['north', 'east']
[2, 1] = ['north']
[2, 2] = ['north', 'west']
[2, 3] = ['north', 'west']
[2, 4] = ['north', 'west']
------------------------------
[3, 0] = ['north', 'east']
[3, 1] = ['north']
[3, 2] = ['north', 'west']
[3, 3] = ['north', 'west']
[3, 4] = ['north', 'west']
------------------------------
[4, 0] = ['north', 'east']
[4, 1] = ['north']
[4, 2] = ['north', 'west']
[4, 3] = ['north', 'west']
[4, 4] = ['north', 'west']
------------------------------
```

As you can see, this pi_* matches what is in the textbook in Figure 3.5. Comments explaining the code are in the code itself.

b.)

```
==============================
==  Optimal State Value    ==
==============================
[[22.  24.4 22.  19.4 17.5]
 [19.8 22.  19.8 17.8 16. ]
 [17.8 19.8 17.8 16.  14.4]
 [16.  17.8 16.  14.4 13. ]
 [14.4 16.  14.4 13.  11.7]]
==============================
==============================
==     Optimal Policy      ==
==============================
[0, 0] = ['east']
[0, 1] = ['north', 'south', 'west', 'east']
[0, 2] = ['west']
[0, 3] = ['north', 'south', 'west', 'east']
[0, 4] = ['west']
------------------------------
[1, 0] = ['north', 'east']
[1, 1] = ['north']
[1, 2] = ['north', 'west']
[1, 3] = ['west']
[1, 4] = ['west']
------------------------------
[2, 0] = ['north', 'east']
[2, 1] = ['north']
[2, 2] = ['north', 'west']
[2, 3] = ['north', 'west']
[2, 4] = ['north', 'west']
------------------------------
[3, 0] = ['north', 'east']
[3, 1] = ['north']
[3, 2] = ['north', 'west']
[3, 3] = ['north', 'west']
[3, 4] = ['north', 'west']
------------------------------
[4, 0] = ['north', 'east']
[4, 1] = ['north']
[4, 2] = ['north', 'west']
[4, 3] = ['north', 'west']
[4, 4] = ['north', 'west']
------------------------------
```

As you can see, this pi_* also matches Figure 3.5 in the textbook. Comments explaining the code are located in the code itself.

5.) a.)

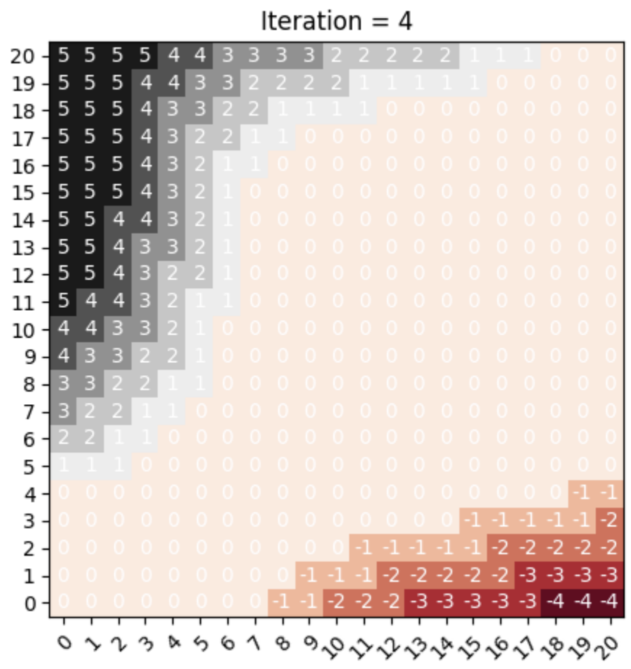


Figure: pi*, found on the 4th iteration

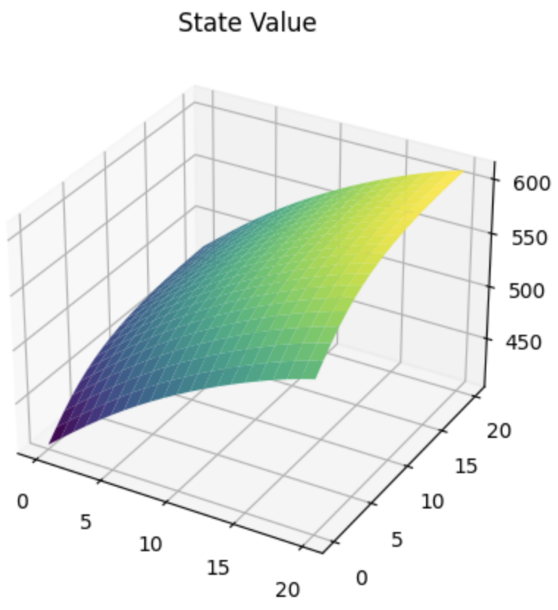
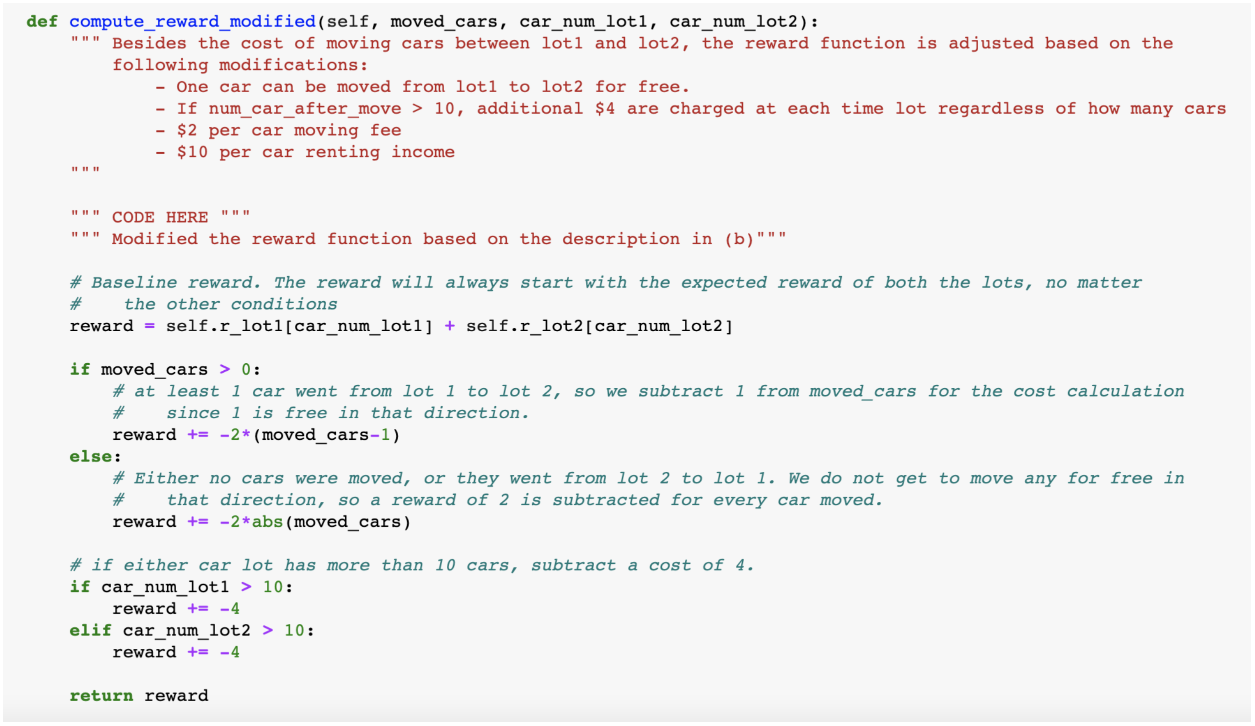


Figure: 3D plot of v_{pi}* for the 4th iteration

5b.) Following the modifications made in part b, I modified the method "compute_reward_modified" as follows:



```python
def compute_reward_modified(self, moved_cars, car_num_lot1, car_num_lot2):
    """ Besides the cost of moving cars between lot1 and lot2, the reward function is adjusted based on the
        following modifications:
            - One car can be moved from lot1 to lot2 for free.
            - If num_car_after_move > 10, additional $4 are charged at each time lot regardless of how many cars
            - $2 per car moving fee
            - $10 per car renting income
    """

    """ CODE HERE """
    """ Modified the reward function based on the description in (b)"""

    # Baseline reward. The reward will always start with the expected reward of both the lots, no matter
    #    the other conditions
    reward = self.r_lot1[car_num_lot1] + self.r_lot2[car_num_lot2]

    if moved_cars > 0:
        # at least 1 car went from lot 1 to lot 2, so we subtract 1 from moved_cars for the cost calculation
        #    since 1 is free in that direction.
        reward += -2*(moved_cars-1)
    else:
        # Either no cars were moved, or they went from lot 2 to lot 1. We do not get to move any for free in
        #    that direction, so a reward of 2 is subtracted for every car moved.
        reward += -2*abs(moved_cars)

    # if either car lot has more than 10 cars, subtract a cost of 4.
    if car_num_lot1 > 10:
        reward += -4
    elif car_num_lot2 > 10:
        reward += -4

    return reward
```

The code has comments that explain the changes I made, although I will explain here as well. The reward calculation begins with the summation of the expected reward of both of the lots, and will always start with that because that will not change no matter the other situations. I then check whether or not cars were moved from lot 1 to lot 2, or the other way around. If moved_cars is positive, that means at least 1 car was moved from lot 1 to lot 2. We then multiply the cost to move each car, -2, with the number of cars moved from lot 1 to lot 2, minus 1, since our coworker can move 1 for free. If there were no cars moved, or they were moved from lot 2 to lot 1 (negative value of moved_cars), our cost is multiplied by the absolute value of the number since we cannot move any of them for free in that direction.

I then check whether the total number of cars in each lot after they are moved is greater than 10. If so, a cost of 4 is subtracted from the reward for each lot that is over 10. The results of this reward function are shown below in 5b.

5b.)

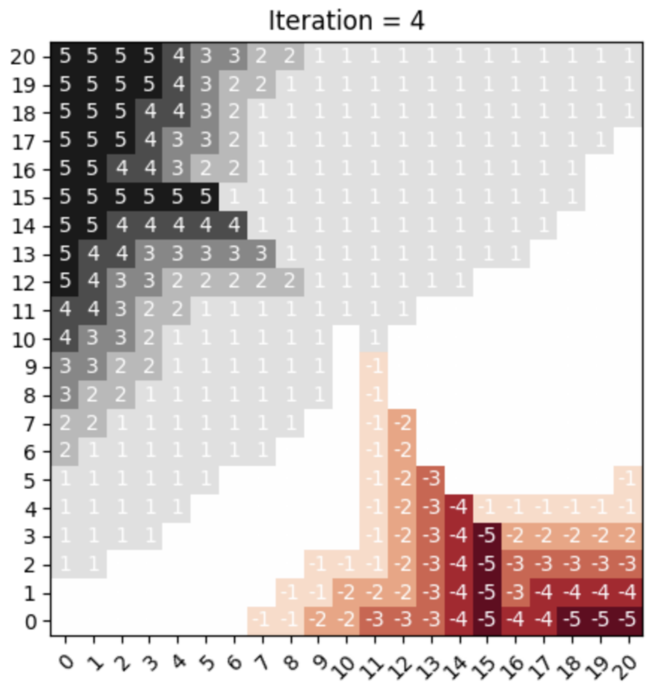


Figure: pi* for the modified reward function, found after the 4th iteration

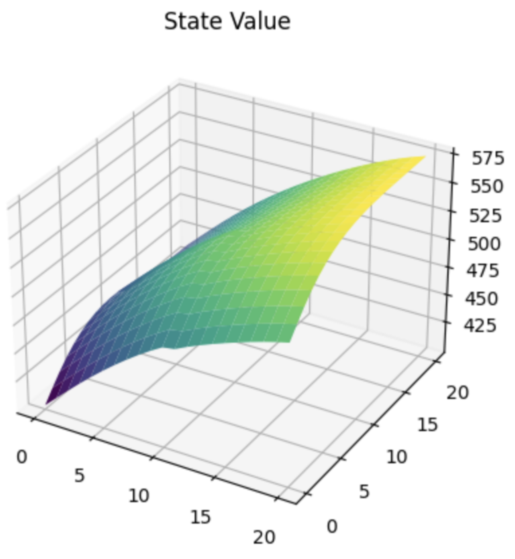


Figure: v_{pi}* for the 4th iteration

The above pi* figure makes sense for a few reasons. First, it is easy to realize that there are a lot more actions of “+1”, or moving 1 car from lot 1 to lot 2, compared to the first reward function. This is because moving 1 car from lot 1 to lot 2 is free thanks to our coworker! It will have no negative reward, therefore being selected much more often. Second, the cost of having more than 10 cars is clearly seen in both lots. In columns 11, 12, 13, 14 and 15, you can see a pattern of actions of -1, -2, -3, -4, and -5, respectively. This is because of the desire to get to 10 or less cars. If there are 11 cars in lot 2, we want to move 1 back to lot 1 (action = -1) to avoid the -4 cost. If there are 12, we want to move 2 back to lot 1 (action = -2) and so on and so forth. The same pattern is seen in rows 11, 12, 13, 14, and 15, with actions of +1, +2, +3, +4, and +5, respectively.