Assignment2

July 2, 2022

1 Assignment 2: Optimal Policies with Dynamic Programming

Welcome to Assignment 2. This notebook will help you understand: - Policy Evaluation and Policy Improvement. - Value and Policy Iteration. - Bellman Equations.

1.1 Gridworld City

Gridworld City, a thriving metropolis with a booming technology industry, has recently experienced an influx of grid-loving software engineers. Unfortunately, the city's street parking system, which charges a fixed rate, is struggling to keep up with the increased demand. To address this, the city council has decided to modify the pricing scheme to better promote social welfare. In general, the city considers social welfare higher when more parking is being used, the exception being that the city prefers that at least one spot is left unoccupied (so that it is available in case someone really needs it). The city council has created a Markov decision process (MDP) to model the demand for parking with a reward function that reflects its preferences. Now the city has hired you — an expert in dynamic programming — to help determine an optimal policy.

1.2 Preliminaries

You'll need two imports to complete this assigment: - numpy: The fundamental package for scientific computing with Python. - tools: A module containing an environment and a plotting function.

There are also some other lines in the cell below that are used for grading and plotting — you needn't worry about them.

In this notebook, all cells are locked except those that you are explicitly asked to modify. It is up to you to decide how to implement your solution in these cells, **but please do not import other libraries** — doing so will break the autograder.

[2]:
```
%matplotlib inline
import numpy as np
import tools
import grader
```

In the city council's parking MDP, states are nonnegative integers indicating how many parking spaces are occupied, actions are nonnegative integers designating the price of street parking, the reward is a real value describing the city's preference for the situation, and time is discretized by hour. As might be expected, charging a high price is likely to decrease occupancy over the hour, while charging a low price is likely to increase it.

For now, let's consider an environment with three parking spaces and three price points. Note that an environment with three parking spaces actually has four states — zero, one, two, or three spaces could be occupied.

[3]:
```
# ---------------
# Discussion Cell
# ---------------
num_spaces = 3
num_prices = 3
env = tools.ParkingWorld(num_spaces, num_prices)
V = np.zeros(num_spaces + 1)
pi = np.ones((num_spaces + 1, num_prices)) / num_prices
```

The value function is a one-dimensional array where the i-th entry gives the value of i spaces being occupied.

[4]:
```
V
```

[4]:
```
array([0., 0., 0., 0.])
```

We can represent the policy as a two-dimensional array where the (i, j)-th entry gives the probability of taking action j in state i.

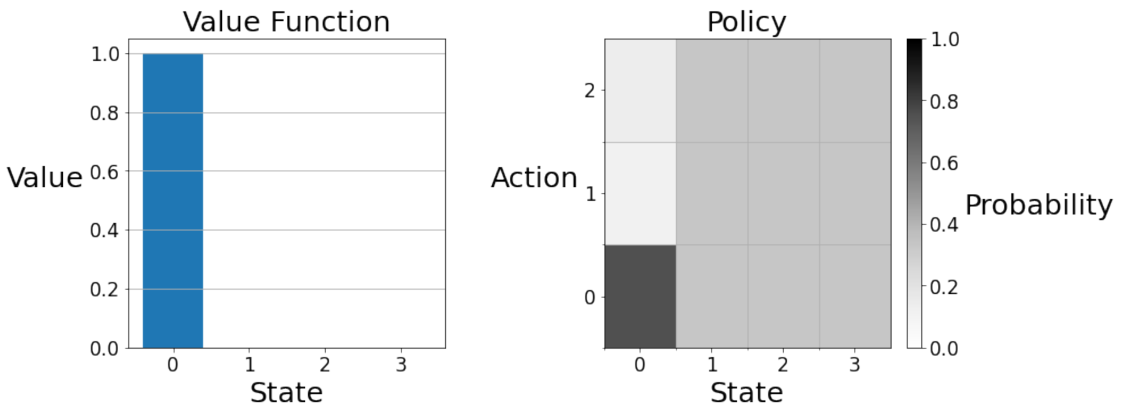
[5]:
```
pi
```

[5]:
```
array([[0.33333333, 0.33333333, 0.33333333],
       [0.33333333, 0.33333333, 0.33333333],
       [0.33333333, 0.33333333, 0.33333333],
       [0.33333333, 0.33333333, 0.33333333]])
```

[6]:
```
pi[0] = [0.75, 0.11, 0.14]

for s, pi_s in enumerate(pi):
    for a, p in enumerate(pi_s):
        print(f'pi(A={a}|S={s}) = {p.round(2)}    ', end='')
    print()
```

```
pi(A=0|S=0) = 0.75    pi(A=1|S=0) = 0.11    pi(A=2|S=0) = 0.14
pi(A=0|S=1) = 0.33    pi(A=1|S=1) = 0.33    pi(A=2|S=1) = 0.33
pi(A=0|S=2) = 0.33    pi(A=1|S=2) = 0.33    pi(A=2|S=2) = 0.33
pi(A=0|S=3) = 0.33    pi(A=1|S=3) = 0.33    pi(A=2|S=3) = 0.33
```

[6]:
```
V[0] = 1

tools.plot(V, pi)
```



We can visualize a value function and policy with the `plot` function in the `tools` module. On the left, the value function is displayed as a barplot. State zero has an expected return of ten, while the other states have an expected return of zero. On the right, the policy is displayed on a two-dimensional grid. Each vertical strip gives the policy at the labeled state. In state zero, action zero is the darkest because the agent's policy makes this choice with the highest probability. In the other states the agent has the equiprobable policy, so the vertical strips are colored uniformly.

You can access the state space and the action set as attributes of the environment.

```
[7]: env.S
```

```
[7]: [0, 1, 2, 3]
```

```
[8]: env.A
```

```
[8]: [0, 1, 2]
```

You will need to use the environment's `transitions` method to complete this assignment. The method takes a state and an action and returns a 2-dimensional array, where the entry at $(i, 0)$ is the reward for transitioning to state i from the current state and the entry at $(i, 1)$ is the conditional probability of transitioning to state i given the current state and action.

```
[9]: state = 3
     action = 1
     transitions = env.transitions(state, action)
     transitions
```

```
[9]: array([[1.        , 0.12390437],
            [2.        , 0.15133714],
            [3.        , 0.1848436 ],
            [2.        , 0.53991488]])
```

```
[10]: for sp, (r, p) in enumerate(transitions):
          print(f'p(S\'={sp}, R={r} | S={state}, A={action}) = {p.round(2)}')
```

```
p(S'=0, R=1.0 | S=3, A=1) = 0.12
p(S'=1, R=2.0 | S=3, A=1) = 0.15
p(S'=2, R=3.0 | S=3, A=1) = 0.18
p(S'=3, R=2.0 | S=3, A=1) = 0.54
```

1.3 Section 1: Policy Evaluation

You're now ready to begin the assignment! First, the city council would like you to evaluate the quality of the existing pricing scheme. Policy evaluation works by iteratively applying the Bellman equation for v_π to a working value function, as an update rule, as shown below.

$$v(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v(s')]$$

This update can either occur "in-place" (i.e. the update rule is sequentially applied to each state) or with "two-arrays" (i.e. the update rule is simultaneously applied to each state). Both versions converge to v_π but the in-place version usually converges faster. **In this assignment, we will be implementing all update rules in-place**, as is done in the pseudocode of chapter 4 of the textbook.

We have written an outline of the policy evaluation algorithm described in chapter 4.1 of the textbook. It is left to you to fill in the `bellman_update` function to complete the algorithm.

[11]:
```python
# lock
def evaluate_policy(env, V, pi, gamma, theta):
    delta = float('inf')
    while delta > theta:
        delta = 0
        for s in env.S:
            v = V[s]
            bellman_update(env, V, pi, s, gamma)
            delta = max(delta, abs(v - V[s]))

    return V
```

[21]:
```python
# -----------
# Graded Cell
# -----------
def bellman_update(env, V, pi, s, gamma):
    """Mutate ``V`` according to the Bellman update equation."""
    # YOUR CODE HERE
#     raise NotImplementedError()

    v = 0
    for idx, act in enumerate(env.A):
        trans = env.transitions(s, act)
        for s_next, (reward,prob) in enumerate(trans):
            v += pi[s][act] * prob * ( reward + gamma * V[s_next] )

    V[s] = v
#     print(f"the new v[{s}] is {v}")


#         return V

```

The cell below uses the policy evaluation algorithm to evaluate the city's policy, which charges a constant price of one.

[18]:
```
# ---------------
# Debugging Cell
# ---------------
# Feel free to make any changes to this cell to debug your code

# set up test environment
num_spaces = 10
num_prices = 4
env = tools.ParkingWorld(num_spaces, num_prices)
print(f"env state = {env.S}")
print(f"env actions are {env.A}")

# build test policy
city_policy = np.zeros((num_spaces + 1, num_prices))
city_policy[:, 1] = 1
print(f"city_policy is {city_policy}")

gamma = 0.9
theta = 0.1

V = np.zeros(num_spaces + 1)
V = evaluate_policy(env, V, city_policy, gamma, theta)

print(V)
```

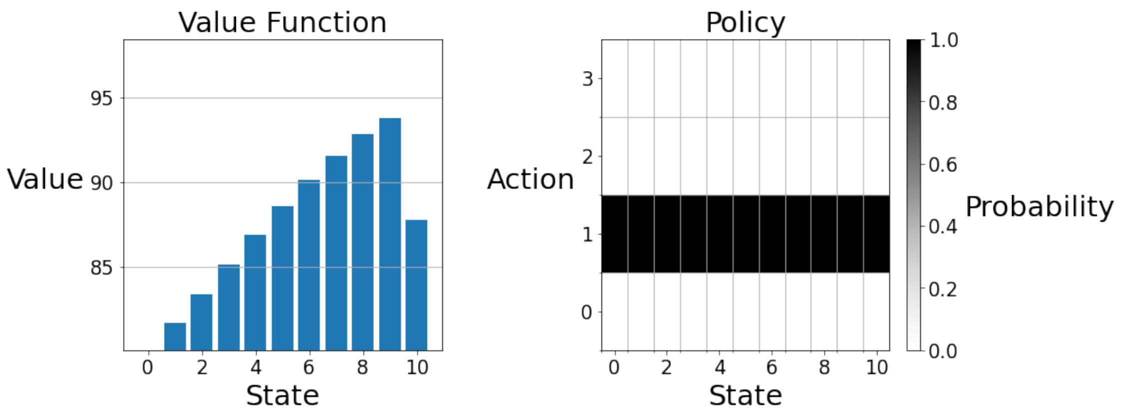
```
env state = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
env actions are [0, 1, 2, 3]
city_policy is [[0. 1. 0. 0.]
 [0. 1. 0. 0.]
 [0. 1. 0. 0.]
 [0. 1. 0. 0.]
 [0. 1. 0. 0.]
 [0. 1. 0. 0.]
 [0. 1. 0. 0.]
 [0. 1. 0. 0.]
 [0. 1. 0. 0.]
 [0. 1. 0. 0.]
 [0. 1. 0. 0.]]
the new v[0] is 3.147637674299018
the new v[1] is 4.689618620824781
the new v[2] is 6.329577316514543
the new v[3] is 8.040812314496803
the new v[4] is 9.803056395762187
the new v[5] is 11.598580090433172
the new v[6] is 13.40998833508625
the new v[7] is 15.218716273480618
the new v[8] is 17.006093150982487
the new v[9] is 18.745717526266073
the new v[10] is 13.737087503356495
the new v[0] is 11.297026962245205
the new v[1] is 13.735484612100674
the new v[2] is 16.16278779868797
the new v[3] is 18.560603252887276
the new v[4] is 20.90658432491496
the new v[5] is 23.17336500296397
the new v[6] is 25.3275809838817
the new v[7] is 27.328650663474974
the new v[8] is 29.122076079193977
the new v[9] is 30.632535965226367
the new v[10] is 25.113716051957603
the new v[0] is 20.61506545783421
the new v[1] is 23.30820160987587
the new v[2] is 25.914406301870002
the new v[3] is 28.430217580731547
the new v[4] is 30.841430957256964
the new v[5] is 33.12603794972545
the new v[6] is 35.25546803198619
the new v[7] is 37.194946797829076
the new v[8] is 38.89558695021129
the new v[9] is 40.29556455873449
the new v[10] is 34.67484837963126
the new v[0] is 29.339431136573168
the new v[1] is 31.994630754623316
the new v[2] is 34.54942609509675
the new v[3] is 37.00466762558306
the new v[4] is 39.34881258948133
the new v[5] is 41.562041434685085
the new v[6] is 43.61796271785444
the new v[7] is 45.4841267873214
the new v[8] is 47.1127469146594
the new v[9] is 48.444831511835346
the new v[10] is 42.76395266975513
the new v[0] is 36.981794790285946
the new v[1] is 39.51410154254195
the new v[2] is 41.962777992773475
the new v[3] is 44.3241246090994
the new v[4] is 46.5838417330371
the new v[5] is 48.72042377129128
the new v[6] is 50.706438705013426
the new v[7] is 52.508774022960004
the new v[8] is 54.078177251729315
the new v[9] is 55.35539229412546
the new v[10] is 49.6246172454149
the new v[0] is 43.53286357968816
the new v[1] is 45.934199803709696
the new v[2] is 48.27655109577442
the new v[3] is 50.54836984249379
the new v[4] is 52.73072957786323
the new v[5] is 54.799226116804455
the new v[6] is 56.7245201497156
the new v[7] is 58.47219031924462
the new v[8] is 59.99122514568727
the new v[9] is 61.22176354960611
the new v[10] is 55.44837024134253
the new v[0] is 49.111518445477635
the new v[1] is 51.3946710247054
the new v[2] is 53.64285421903807
the new v[3] is 55.83645691383945
the new v[4] is 57.95189504658481
the new v[5] is 59.961888170469166
the new v[6] is 61.83522888751815
the new v[7] is 63.536235270517786
the new v[8] is 65.01232234083349
the new v[9] is 66.20307536666253
the new v[10] is 60.39336481906423
the new v[0] is 53.85296581757177
the new v[1] is 56.03402649698669
the new v[2] is 58.20132042733386
the new v[3] is 60.327982606472844
the new v[4] is 62.38628192867881
the new v[5] is 64.34639774001306
the new v[6] is 66.17548483711226
the new v[7] is 67.83676419931618
the new v[8] is 69.27630272997884
the new v[9] is 70.43321091879535
the new v[10] is 64.59261738694437
the new v[0] is 57.88059383834537
the new v[1] is 59.97452588832333
the new v[2] is 62.072896582034296
the new v[3] is 64.14257510602646
the new v[4] is 66.15226265989946
the new v[5] is 68.06995983094657
the new v[6] is 69.86142003001356
the new v[7] is 71.48892737369702
the new v[8] is 72.89740154231836
the new v[9] is 74.02554870061724
the new v[10] is 68.15871562064316
the new v[0] is 61.301290123044964
the new v[1] is 63.321118508001604
the new v[2] is 65.36089531069896
the new v[3] is 67.382142413964
the new v[4] is 69.35052038690961
the new v[5] is 71.2321746170092
the new v[6] is 72.99166600129878
the new v[7] is 74.59048177563946
the new v[8] is 75.9725666809162
the new v[9] is 77.07628309732914
the new v[10] is 71.18716259424016
the new v[0] is 64.20635822819912
the new v[1] is 66.16322463772603
the new v[2] is 68.15322353425051
the new v[3] is 70.13332915579994
the new v[4] is 72.06661718073792
the new v[5] is 73.91765624951427
the new v[6] is 75.64999388895666
the new v[7] is 77.2244402239374
the new v[8] is 78.58411181819855
the new v[9] is 79.66707894776584
the new v[10] is 73.75902995674107
the new v[0] is 66.67347817606036
the new v[1] is 68.57686612899559
the new v[2] is 70.5245864072413
the new v[3] is 72.46974952004729
the new v[4] is 74.37323524403593
the new v[5] is 76.19827283891316
the new v[6] is 77.90754904857442
the new v[7] is 79.46129885349241
the new v[8] is 80.80193542518961
the new v[9] is 81.86728092408188
the new v[10] is 75.9431567813592
the new v[0] is 68.76865880133533
the new v[1] is 70.62662819101648
the new v[2] is 72.5384422209936
the new v[3] is 74.45392970632558
the new v[4] is 76.33210537530132
the new v[5] is 78.13506093074074
the new v[6] is 79.82475204797291
the new v[7] is 81.3609252362258
the new v[8] is 82.68539629625198
the new v[9] is 83.73577666344221
the new v[10] is 77.79800077549507
the new v[0] is 70.54796911805522
the new v[1] is 72.36736652646815
the new v[2] is 74.24868709113824
the new v[3] is 76.13897255083307
the new v[4] is 77.9956537020356
the new v[5] is 79.77985615887796
the new v[6] is 81.45291472692297
the new v[7] is 82.97416107541446
the new v[8] is 84.28490368766438
the new v[9] is 85.32257503838711
the new v[10] is 79.37320552978946
the new v[0] is 72.05902864283922
the new v[1] is 73.8456689983258
the new v[2] is 75.70109318537764
the new v[3] is 77.56997600983726
the new v[4] is 79.4084031137303
the new v[5] is 81.17667966173971
the new v[6] is 82.83561318728182
the new v[7] is 84.34418305798376
the new v[8] is 85.64326692143487
the new v[9] is 86.67014527055338
the new v[10] is 80.71093000277764
the new v[0] is 73.34227878659495
the new v[1] is 75.10110051741282
the new v[2] is 76.93453245650528
the new v[3] is 78.78523930685088
the new v[4] is 80.6081643478111
the new v[5] is 82.36291598380312
the new v[6] is 84.0098539666956
the new v[7] is 85.50765847493022
the new v[8] is 86.79684127312959
the new v[9] is 87.8145537849584
the new v[10] is 81.84697711627643
the new v[0] is 74.43206428498148
the new v[1] is 76.16726134585056
the new v[2] is 77.98201660674526
the new v[3] is 79.81728771013238
the new v[4] is 81.62704779525853
the new v[5] is 83.3703135499704
the new v[6] is 85.0070644522698
the new v[7] is 86.49572659728777
the new v[8] is 87.7765010275339
the new v[9] is 88.78642955595201
the new v[10] is 82.81175206271467
the new v[0] is 75.35755211200393
the new v[1] is 77.07268618651122
the new v[2] is 78.87158048871092
the new v[3] is 80.69374295839914
the new v[4] is 82.49232285394262
the new v[5] is 84.22583435440858
the new v[6] is 85.85393399442422
the new v[7] is 87.33483209168728
the new v[8] is 88.60846581146049
the new v[9] is 89.61178388099958
the new v[10] is 83.63107609384275
the new v[0] is 76.14351201639599
the new v[1] is 77.84160782822326
the new v[2] is 79.62703239221861
the new v[3] is 81.43806250449536
the new v[4] is 83.22714775188827
the new v[5] is 84.9523755635726
the new v[6] is 86.57312821817678
the new v[7] is 88.04743278686212
the new v[8] is 89.3150023401532
the new v[9] is 90.31270655334183
the new v[10] is 84.32687760830467
the new v[0] is 76.81097943215865
the new v[1] is 78.49460569460818
the new v[2] is 80.26859123785582
the new v[3] is 82.07016732328584
the new v[4] is 83.85118934967647
the new v[5] is 85.56938233373626
the new v[6] is 87.18389564532376
the new v[7] is 88.65260073557609
the new v[8] is 89.91502036513134
the new v[9] is 90.90795707530575
the new v[10] is 84.91777904578095
the new v[0] is 77.37781846276266
the new v[1] is 79.04915662653438
the new v[2] is 80.81342771277997
the new v[3] is 82.60697507450048
the new v[4] is 84.38114950308201
the new v[5] is 86.09336824060347
the new v[6] is 87.70258286182329
the new v[7] is 89.16653265935895
the new v[8] is 90.42457877536336
the new v[9] is 91.41346673839439
the new v[10] is 85.41959529885452
the new v[0] is 77.85919999363318
the new v[1] is 79.52010263079744
the new v[2] is 81.27612382606634
the new v[3] is 83.06285288644307
the new v[4] is 84.83121207128181
the new v[5] is 86.5383572485928
the new v[6] is 88.14307201791785
the new v[7] is 89.60298343777005
the new v[8] is 90.857315397661
the new v[9] is 91.84276500871857
the new v[10] is 85.84575698317803
the new v[0] is 78.26800772510072
the new v[1] is 79.92004811123519
the new v[2] is 81.66906318141592
the new v[3] is 83.45000187774737
the new v[4] is 85.21342253368023
the new v[5] is 86.91625904830073
the new v[6] is 88.51715237012857
the new v[7] is 89.97363424415347
the new v[8] is 91.22481199965905
the new v[9] is 92.2073416298272
the new v[10] is 86.20766989468117
the new v[0] is 78.61518299165303
the new v[1] is 80.25969721296985
the new v[2] is 82.00276241195405
the new v[3] is 83.7787837081801
the new v[4] is 85.53801037520824
the new v[5] is 87.23718780756738
the new v[6] is 88.83483580903632
the new v[7] is 90.28840518069067
the new v[8] is 91.53690426438033
the new v[9] is 92.51695413434413
the new v[10] is 86.51502027443584
the new v[0] is 78.91001759585507
the new v[1] is 80.54814030821504
the new v[2] is 82.28615264748066
the new v[3] is 84.05799789819396
the new v[4] is 85.81366286835583
the new v[5] is 87.50973286716675
the new v[6] is 89.10462481762687
the new v[7] is 90.55572078012324
the new v[8] is 91.80194503256547
the new v[9] is 92.77988899468151
the new v[10] is 86.77603405132282
the new v[0] is 79.16040249422998
the new v[1] is 80.79309729108084
the new v[2] is 82.52681854731482
the new v[3] is 84.29511733938155
the new v[4] is 86.04775757942276
the new v[5] is 87.74118862601392
the new v[6] is 89.33374003183665
the new v[7] is 90.78273548003838
the new v[8] is 92.02702785819383
the new v[9] is 93.00318340227838
the new v[10] is 86.99769700088308
the new v[0] is 79.37303899089783
the new v[1] is 81.00112419289853
the new v[2] is 82.73120129619554
the new v[3] is 84.49648829902458
the new v[4] is 86.24655982178128
the new v[5] is 87.93774976853564
the new v[6] is 89.52831349373056
the new v[7] is 90.97552510431385
the new v[8] is 92.21817686043889
the new v[9] is 93.19281361106879
the new v[10] is 87.18594171269018
the new v[0] is 79.55361809134901
the new v[1] is 81.17778864791237
the new v[2] is 82.90477099621864
the new v[3] is 84.66750027165932
the new v[4] is 86.4153903407906
the new v[5] is 88.10467705927091
the new v[6] is 89.69355276954187
the new v[7] is 91.13924947653359
the new v[8] is 92.38050795325898
the new v[9] is 93.35385488600667
the new v[10] is 87.34580637024183
the new v[0] is 79.7069728165761
the new v[1] is 81.32781890550218
the new v[2] is 83.0521730687081
the new v[3] is 84.81273022356632
the new v[4] is 86.55876771828291
the new v[5] is 88.24643814202986
the new v[6] is 89.83388032503107
the new v[7] is 91.27829051756214
the new v[8] is 92.51836576777366
the new v[9] is 93.49061733751188
the new v[10] is 87.48156959295748
the new v[0] is 79.83720755409946
the new v[1] is 81.45523037710076
the new v[2] is 83.17735258432222
the new v[3] is 84.93606509074525
the new v[4] is 86.68052930793625
the new v[5] is 88.36682711196089
the new v[6] is 89.95305188840724
the new v[7] is 91.39636952332161
the new v[8] is 92.63543993201789
the new v[9] is 93.60676127731196
the new v[10] is 87.59686494919458
the new v[0] is 79.94780790783923
the new v[1] is 81.5634331047085
the new v[2] is 83.28365984896743
the new v[3] is 85.04080580891183
the new v[4] is 86.7839339382497
the new v[5] is 88.469066060754
the new v[6] is 90.05425696867825
the new v[7] is 91.496646761596
the new v[8] is 92.73486382018655
the new v[9] is 93.70539518306475
the new v[10] is 87.69477820511132
the new v[0] is 80.0417339868561
the new v[1] is 81.65532302729446
the new v[2] is 83.37394007142157
the new v[3] is 85.12975565977712
the new v[4] is 86.87174913186462
the new v[5] is 88.55589131273774
the new v[6] is 90.14020421970432
the new v[7] is 91.58180605347238
the new v[8] is 92.81929841429525
the new v[9] is 93.78915889368793
the new v[10] is 87.7779299121454
[80.04173399 81.65532303 83.37394007 85.12975566 86.87174913 88.55589131
 90.14020422 91.58180605 92.81929841 93.78915889 87.77792991]
```

[22]:
```
# -----------
# Tested Cell
# -----------
# The contents of the cell will be tested by the autograder.
# If they do not pass here, they will not pass there.

# set up test environment
num_spaces = 10
num_prices = 4
env = tools.ParkingWorld(num_spaces, num_prices)

# build test policy
city_policy = np.zeros((num_spaces + 1, num_prices))
city_policy[:, 1] = 1

gamma = 0.9
theta = 0.1

V = np.zeros(num_spaces + 1)
V = evaluate_policy(env, V, city_policy, gamma, theta)

# test the value function
answer = [80.04, 81.65, 83.37, 85.12, 86.87, 88.55, 90.14, 91.58, 92.81, 93.78,␣
 ↪87.77]
```

```
# make sure the value function is within 2 decimal places of the correct answer
assert grader.near(V, answer, 1e-2)
```

You can use the `plot` function to visualize the final value function and policy.

```
[23]: # lock
tools.plot(V, city_policy)
```



Observe that the value function qualitatively resembles the city council's preferences — it monotonically increases as more parking is used, until there is no parking left, in which case the value is lower. Because of the relatively simple reward function (more reward is accrued when many but not all parking spots are taken and less reward is accrued when few or all parking spots are taken) and the highly stochastic dynamics function (each state has positive probability of being reached each time step) the value functions of most policies will qualitatively resemble this graph. However, depending on the intelligence of the policy, the scale of the graph will differ. In other words, better policies will increase the expected return at every state rather than changing the relative desirability of the states. Intuitively, the value of a less desirable state can be increased by making it less likely to remain in a less desirable state. Similarly, the value of a more desirable state can be increased by making it more likely to remain in a more desirable state. That is to say, good policies are policies that spend more time in desirable states and less time in undesirable states. As we will see in this assignment, such a steady state distribution is achieved by setting the price to be low in low occupancy states (so that the occupancy will increase) and setting the price high when occupancy is high (so that full occupancy will be avoided).

1.4 Section 2: Policy Iteration

Now the city council would like you to compute a more efficient policy using policy iteration. Policy iteration works by alternating between evaluating the existing policy and making the policy greedy with respect to the existing value function. We have written an outline of the policy iteration algorithm described in chapter 4.3 of the textbook. We will make use of the policy evaluation algorithm you completed in section 1. It is left to you to fill in the `q_greedify_policy` function, such that it modifies the policy at s to be greedy with respect to the q-values at s, to complete the policy improvement algorithm.

[35]:
```
def improve_policy(env, V, pi, gamma):
    policy_stable = True
    for s in env.S:
        old = pi[s].copy()
        q_greedify_policy(env, V, pi, s, gamma)

        if not np.array_equal(pi[s], old):
            policy_stable = False

    return pi, policy_stable

def policy_iteration(env, gamma, theta):
    V = np.zeros(len(env.S))
    pi = np.ones((len(env.S), len(env.A))) / len(env.A)
    policy_stable = False

    while not policy_stable:
        V = evaluate_policy(env, V, pi, gamma, theta)
        pi, policy_stable = improve_policy(env, V, pi, gamma)

    return V, pi
```

[38]:
```
# -----------
# Graded Cell
# -----------
def q_greedify_policy(env, V, pi, s, gamma):
    """Mutate ``pi`` to be greedy with respect to the q-values induced by ``V``.
 ↪"""
    # YOUR CODE HERE
#     raise NotImplementedError()

    q = np.zeros(len(env.A))

    for act in env.A:
        trans = env.transitions(s, act)

        for s_next, (reward, prob) in enumerate(trans):
            q[act] += prob * (reward + (gamma * V[s_next]))

#     print(f"pi before changes {pi}")
    pi[s, : ] = 0
    pi[s, np.argmax(q)] = 1
#     print(f"pi after changes {pi}")
```

[37]:
```python
# ---------------
# Debugging Cell
# ---------------
# Feel free to make any changes to this cell to debug your code

gamma = 0.9
theta = 0.1
env = tools.ParkingWorld(num_spaces=6, num_prices=4)

V = np.array([7, 6, 5, 4, 3, 2, 1])
pi = np.ones((7, 4)) / 4

new_pi, stable = improve_policy(env, V, pi, gamma)

# expect first call to greedify policy
expected_pi = np.array([
    [0, 0, 0, 1],
    [0, 0, 0, 1],
    [0, 0, 0, 1],
    [0, 0, 0, 1],
    [0, 0, 0, 1],
    [0, 0, 0, 1],
    [0, 0, 0, 1],
])
assert np.all(new_pi == expected_pi)
assert stable == False

# the value function has not changed, so the greedy policy should not change
new_pi, stable = improve_policy(env, V, new_pi, gamma)

assert np.all(new_pi == expected_pi)
assert stable == True
```

```
pi before changes [[0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]]
pi after changes [[0.   0.   0.   1.  ]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]]
```

```
pi before changes [[0.   0.   0.   1.  ]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]]
pi after changes [[0.   0.   0.   1.  ]
 [0.   0.   0.   1.  ]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]]
pi before changes [[0.   0.   0.   1.  ]
 [0.   0.   0.   1.  ]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]]
pi after changes [[0.   0.   0.   1.  ]
 [0.   0.   0.   1.  ]
 [0.   0.   0.   1.  ]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]]
pi before changes [[0.   0.   0.   1.  ]
 [0.   0.   0.   1.  ]
 [0.   0.   0.   1.  ]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]]
pi after changes [[0.   0.   0.   1.  ]
 [0.   0.   0.   1.  ]
 [0.   0.   0.   1.  ]
 [0.   0.   0.   1.  ]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]]
pi before changes [[0.   0.   0.   1.  ]
 [0.   0.   0.   1.  ]
 [0.   0.   0.   1.  ]
 [0.   0.   0.   1.  ]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]]
pi after changes [[0.   0.   0.   1.  ]
 [0.   0.   0.   1.  ]
 [0.   0.   0.   1.  ]
 [0.   0.   0.   1.  ]
 [0.   0.   0.   1.  ]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]]
pi before changes [[0.   0.   0.   1.  ]
 [0.   0.   0.   1.  ]
 [0.   0.   0.   1.  ]
 [0.   0.   0.   1.  ]
 [0.   0.   0.   1.  ]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]]
pi after changes [[0.   0.   0.   1.  ]
 [0.   0.   0.   1.  ]
 [0.   0.   0.   1.  ]
 [0.   0.   0.   1.  ]
 [0.   0.   0.   1.  ]
 [0.   0.   0.   1.  ]
 [0.25 0.25 0.25 0.25]]
pi before changes [[0.   0.   0.   1.  ]
 [0.   0.   0.   1.  ]
 [0.   0.   0.   1.  ]
 [0.   0.   0.   1.  ]
 [0.   0.   0.   1.  ]
 [0.   0.   0.   1.  ]
 [0.25 0.25 0.25 0.25]]
pi after changes [[0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]]
pi before changes [[0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]]
pi after changes [[0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]]
pi before changes [[0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]]
pi after changes [[0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]]
pi before changes [[0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]]
pi after changes [[0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]]
pi before changes [[0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]]
pi after changes [[0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]]
pi before changes [[0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]]
pi after changes [[0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]]
pi before changes [[0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]]
pi after changes [[0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]]
pi before changes [[0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]]
pi after changes [[0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]]
```

[39]:

```
# -----------
# Tested Cell
# -----------
# The contents of the cell will be tested by the autograder.
# If they do not pass here, they will not pass there.
gamma = 0.9
theta = 0.1
env = tools.ParkingWorld(num_spaces=10, num_prices=4)
```

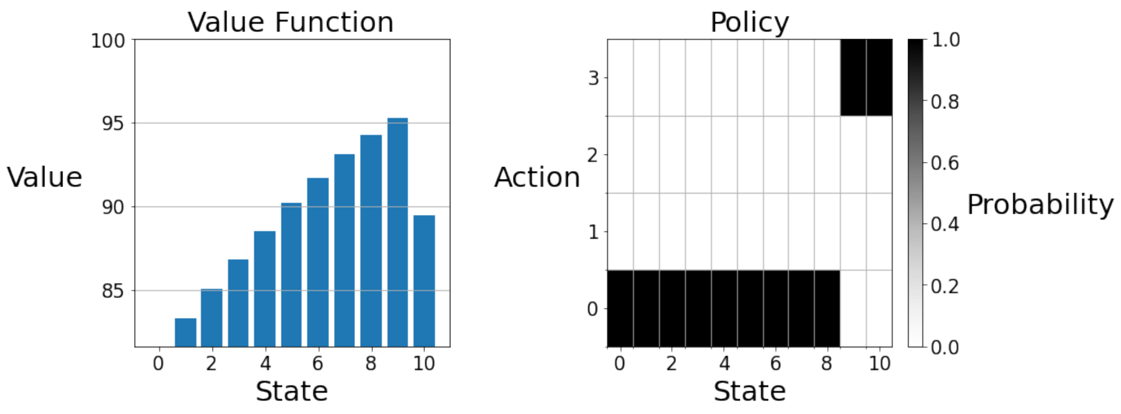
```
V, pi = policy_iteration(env, gamma, theta)

V_answer = [81.60, 83.28, 85.03, 86.79, 88.51, 90.16, 91.70, 93.08, 94.25, 95.
↪25, 89.45]
pi_answer = [
    [1, 0, 0, 0],
    [1, 0, 0, 0],
    [1, 0, 0, 0],
    [1, 0, 0, 0],
    [1, 0, 0, 0],
    [1, 0, 0, 0],
    [1, 0, 0, 0],
    [1, 0, 0, 0],
    [1, 0, 0, 0],
    [0, 0, 0, 1],
    [0, 0, 0, 1],
]

# make sure value function is within 2 decimal places of answer
assert grader.near(V, V_answer, 1e-2)
# make sure policy is exactly correct
assert np.all(pi == pi_answer)
```

When you are ready to test the policy iteration algorithm, run the cell below.

[40]:
```
env = tools.ParkingWorld(num_spaces=10, num_prices=4)
gamma = 0.9
theta = 0.1
V, pi = policy_iteration(env, gamma, theta)
```

You can use the `plot` function to visualize the final value function and policy.

[41]:
```
tools.plot(V, pi)
```



You can check the value function (rounded to one decimal place) and policy against the answer below: State Value Action 0 81.6 0 1 83.3 0 2 85.0 0 3 86.8 0 4 88.5 0 5 90.2 0 6 91.7 0 7 93.1 0 8 94.3 0 9 95.3 3 10 89.5 3

1.5 Section 3: Value Iteration

The city has also heard about value iteration and would like you to implement it. Value iteration works by iteratively applying the Bellman optimality equation for v_* to a working value function, as an update rule, as shown below.

$$v(s) \leftarrow \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma v(s')]$$

We have written an outline of the value iteration algorithm described in chapter 4.4 of the textbook. It is left to you to fill in the `bellman_optimality_update` function to complete the value iteration algorithm.

[42]:
```python
def value_iteration(env, gamma, theta):
    V = np.zeros(len(env.S))
    while True:
        delta = 0
        for s in env.S:
            v = V[s]
            bellman_optimality_update(env, V, s, gamma)
            delta = max(delta, abs(v - V[s]))
        if delta < theta:
            break
    pi = np.ones((len(env.S), len(env.A))) / len(env.A)
    for s in env.S:
        q_greedify_policy(env, V, pi, s, gamma)
    return V, pi
```

[56]:
```python
# -----------
# Graded Cell
# -----------
def bellman_optimality_update(env, V, s, gamma):
    """Mutate ``V`` according to the Bellman optimality update equation."""
    # YOUR CODE HERE
#     raise NotImplementedError()

    V_max = -1 * float('INF')

    for act in env.A:
        trans = env.transitions(s, act)
        v = 0
        for s_next, (reward, prob) in enumerate(trans):
            v += prob * (reward + gamma * V[s_next])

#         print(f"In state {s}, the value of v is {v}")
        V_max = max(v, V_max)

    V[s] = V_max
#     print(f"in state {s} the V is {V}")
```

[54]:
```
# --------------
# Debugging Cell
# --------------
# Feel free to make any changes to this cell to debug your code

gamma = 0.9
env = tools.ParkingWorld(num_spaces=6, num_prices=4)

V = np.array([7, 6, 5, 4, 3, 2, 1])

# only state 0 updated
bellman_optimality_update(env, V, 0, gamma)
assert list(V) == [5, 6, 5, 4, 3, 2, 1]

# only state 2 updated
bellman_optimality_update(env, V, 2, gamma)
assert list(V) == [5, 6, 7, 4, 3, 2, 1]
```

```
In state 0, the value of v is 5.632981308044116
In state 0, the value of v is 5.639669050758412
In state 0, the value of v is 5.646625084474893
In state 0, the value of v is 5.653852604318307
in state 0 the V is [5 6 5 4 3 2 1]
In state 2, the value of v is 7.300305000148783
In state 2, the value of v is 7.313986140125924
In state 2, the value of v is 7.327875688481355
In state 2, the value of v is 7.338233169882892
in state 2 the V is [5 6 7 4 3 2 1]
```

[57]:
```
# -----------
# Tested Cell
# -----------
# The contents of the cell will be tested by the autograder.
# If they do not pass here, they will not pass there.
gamma = 0.9
env = tools.ParkingWorld(num_spaces=10, num_prices=4)
```

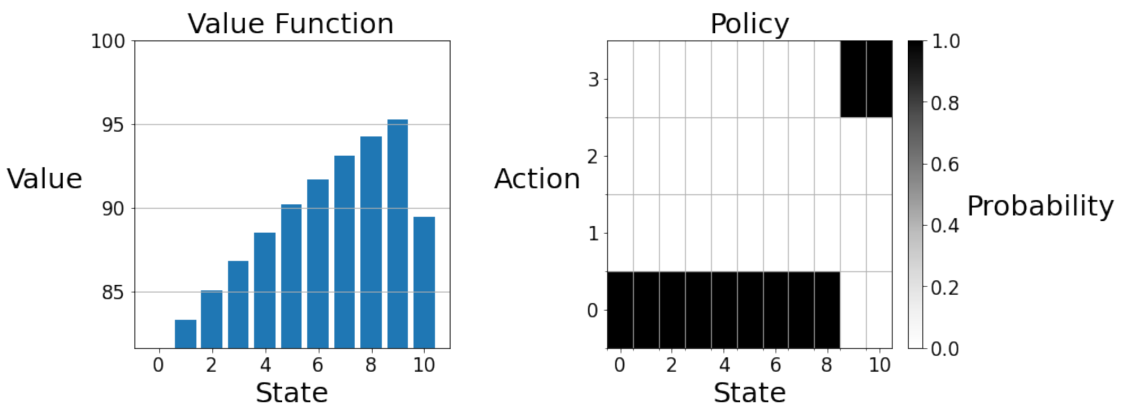
```
V = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])

for _ in range(10):
    for s in env.S:
        bellman_optimality_update(env, V, s, gamma)

# make sure value function is exactly correct
answer = [61, 63, 65, 67, 69, 71, 72, 74, 75, 76, 71]
assert np.all(V == answer)
```

When you are ready to test the value iteration algorithm, run the cell below.

[58]:
```
env = tools.ParkingWorld(num_spaces=10, num_prices=4)
gamma = 0.9
theta = 0.1
V, pi = value_iteration(env, gamma, theta)
```

You can use the `plot` function to visualize the final value function and policy.

[59]:
```
tools.plot(V, pi)
```



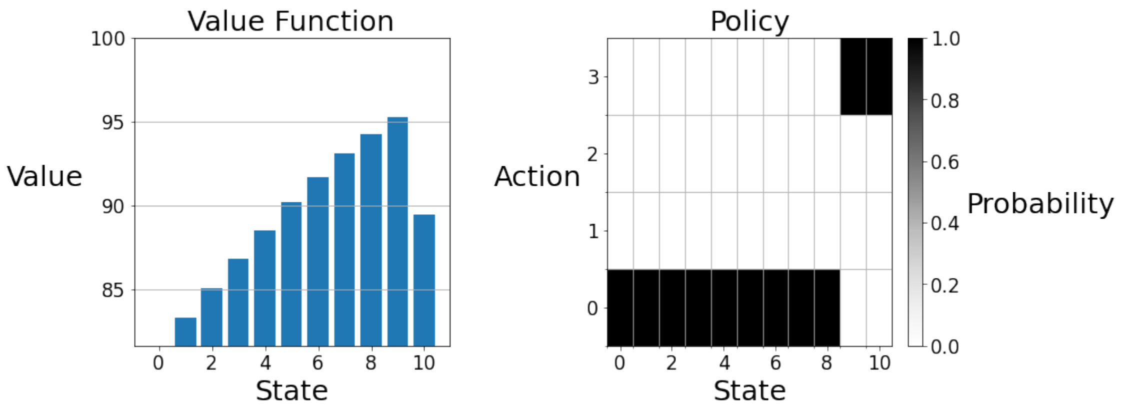
You can check your value function (rounded to one decimal place) and policy against the answer below: State Value Action 0 81.6 0 1 83.3 0 2 85.0 0 3 86.8 0 4 88.5 0 5 90.2 0 6 91.7 0 7 93.1 0 8 94.3 0 9 95.3 3 10 89.5 3

In the value iteration algorithm above, a policy is not explicitly maintained until the value function has converged. Below, we have written an identically behaving value iteration algorithm that maintains an updated policy. Writing value iteration in this form makes its relationship to policy iteration more evident. Policy iteration alternates between doing complete greedifications and complete evaluations. On the other hand, value iteration alternates between doing local greedifications and local evaluations.

[60]:
```
def value_iteration2(env, gamma, theta):
    V = np.zeros(len(env.S))
    pi = np.ones((len(env.S), len(env.A))) / len(env.A)
    while True:
        delta = 0
        for s in env.S:
            v = V[s]
            q_greedify_policy(env, V, pi, s, gamma)
            bellman_update(env, V, pi, s, gamma)
            delta = max(delta, abs(v - V[s]))
        if delta < theta:
            break
    return V, pi
```

You can try the second value iteration algorithm by running the cell below.

[61]:
```
env = tools.ParkingWorld(num_spaces=10, num_prices=4)
gamma = 0.9
theta = 0.1
V, pi = value_iteration2(env, gamma, theta)
tools.plot(V, pi)
```



1.6 Wrapping Up

Congratulations, you've completed assignment 2! In this assignment, we investigated policy evaluation and policy improvement, policy iteration and value iteration, and Bellman updates. Gridworld City thanks you for your service!