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.

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.

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.

In [1]:
```
from google.colab import drive
drive.mount('/content/drive')
import sys
sys.path.append('/content/drive/MyDrive/DSBA M2/2 Reinforcement Learning/TP02_Dynamic_Programming')
```

```
Mounted at /content/drive
```

In [2]:
```
%%capture
%matplotlib inline
import numpy as np
import pickle
import tools
```

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.

In [3]:
```
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
```

In [7]:
```
pi
```

Out[7]:
```
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]])
```

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

In [4]:
```
V
```

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

In [5]:
```
state = 0
V[state]
```

Out[5]:
```
0.0
```

In [6]:
```
state = 0
value = 10
V[state] = value
V
```

Out[6]:
```
array([10.,  0.,  0.,  0.])
```

In [8]:
```
for s, v in enumerate(V):
    print(f'State {s} has value {v}')
```

```
State 0 has value 10.0
State 1 has value 0.0
State 2 has value 0.0
State 3 has value 0.0
```

The policy is a two-dimensional array where the (i, j)-th entry gives the probability of taking action j in state i.

In [9]:
```
pi
```

Out[9]:
```
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]])
```

In [10]:
```
state = 0
pi[state]
```

Out[10]:
```
array([0.33333333, 0.33333333, 0.33333333])
```

In [11]:
```
state = 0
action = 1
pi[state, action]
```

Out[11]:
```
0.3333333333333333
```

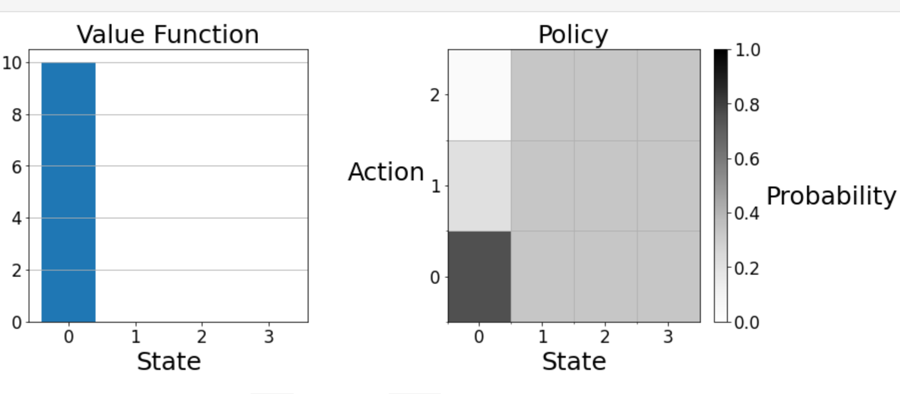
In [12]:
```
pi[state] = np.array([0.75, 0.21, 0.04])
pi
```

Out[12]:
```
array([[0.75      , 0.21      , 0.04      ],
       [0.33333333, 0.33333333, 0.33333333],
       [0.33333333, 0.33333333, 0.33333333],
       [0.33333333, 0.33333333, 0.33333333]])
```

In [13]:
```
for s, pi_s in enumerate(pi):
    print(f''.join(f'pi(A={a}|S={s}) = {p.round(2)}' + 4 * ' ' for a, p in enumerate(pi_s)))
```

```
pi(A=0|S=0) = 0.75    pi(A=1|S=0) = 0.21    pi(A=2|S=0) = 0.04
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
```

In [14]:
```
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.

In [15]:
```
env.S
```

Out[15]:
```
[0, 1, 2, 3]
```

In [16]:
```
env.A
```

Out[16]:
```
[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.

In [23]:
```
state = 3
action = 1
transitions = env.transitions(state, action)
transitions
```

Out[23]:
```
array([[1.        , 0.12390437],
       [2.        , 0.15133714],
       [3.        , 0.1848436 ],
       [2.        , 0.53991488]])
```

In [24]:
```
for s_, (r, p) in enumerate(transitions):
    print(f'p(S\'={s_}, 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
```

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.

In [27]:
```
def evaluate_policy(env, V, pi, gamma, theta):
    while True:
        delta = 0
        for s in env.S:
            v = V[s]
            bellman_update(env, V, pi, s, gamma)
            delta = max(delta, abs(v - V[s]))
        if delta < theta:
            break
    return V
```

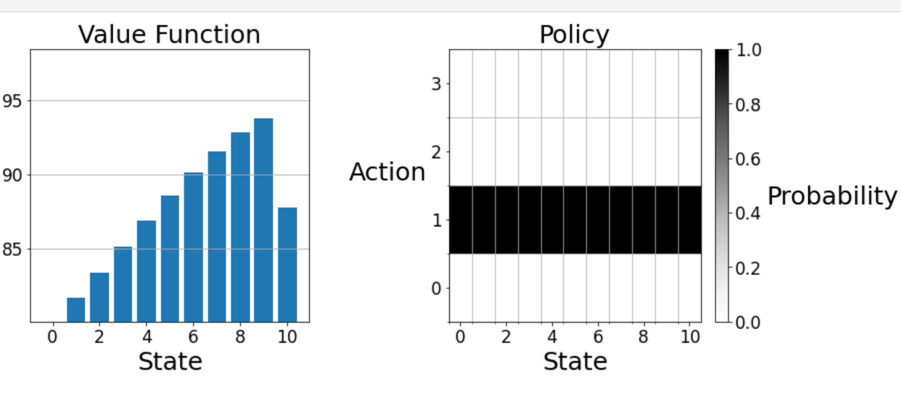
In [26]:
```
# [Graded]
def bellman_update(env, V, pi, s, gamma):
    """
    Mutate ``V`` according to the Bellman update equation.
    """
    ### START CODE HERE ###
    sum1 = np.zeros(len(env.A)) # number of actions in the environment initiated at zero
    for action in env.A:
        transitions = env.transitions(s, action) # given a state, apply transitions for all of the actions
      # for each transition and its probabilities, update the sum for each action
        for s1, (reward, prob) in enumerate(transitions):
            sum1[action] += prob * (reward + gamma * V[s1])
    V[s] = np.sum(pi[s] * sum1)

    ### END CODE HERE ###
```

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

In [28]:
```
%reset_selective -f "^num_spaces$|^num_prices$|^env$|^V$|^pi$|^gamma$|^theta$"
num_spaces = 10
num_prices = 4
env = tools.ParkingWorld(num_spaces, num_prices)
V = np.zeros(num_spaces + 1)
city_policy = np.zeros((num_spaces + 1, num_prices))
city_policy[:, 1] = 1
gamma = 0.9
theta = 0.1
V = evaluate_policy(env, V, city_policy, gamma, theta)
```

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

In [29]:
```
tools.plot(V, city_policy)
```



In [30]:
```
for s, v in enumerate(V):
    print("{}: {:1f}".format(s, v))
```

```
0: 80.041734
1: 81.655323
2: 83.373940
3: 85.129756
4: 86.871749
5: 88.555891
6: 90.140204
7: 91.581806
8: 92.819298
9: 93.789159
10: 87.777930
```

You can check the output (rounded to one decimal place) against the answer below:

State	Value
0	80.0
1	81.7
2	83.4
3	85.1
4	86.9
5	88.6
6	90.1
7	91.6
8	92.8
9	93.8
10	87.8

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).

The cell below will check that your code passes the test case above. (Your code passed if the cell runs without error.)

In [31]:
```
V_correct = [80.0,81.7,83.4,85.1,86.9,88.6,90.1,91.6,92.8,93.8,87.8]
np.testing.assert_array_almost_equal(V, V_correct,1)
print("Assertion correct. You can now move to the next section.")
```

```
Assertion correct. You can now move to the next section.
```

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.

In [32]:
```
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
```

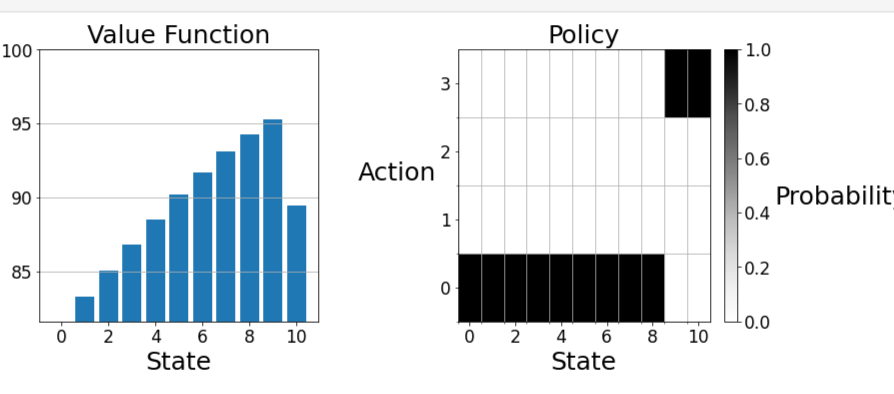
In [33]:
```
# [Graded]
def q_greedify_policy(env, V, pi, s, gamma):
    """
    Mutate ``pi`` to be greedy with respect to the q-values induced by ``V``.
    """
    ### START CODE HERE ###
    sum1 = np.zeros(len(env.A)) # number of actions in the environment initiated at zero
    for action in env.A:
        transitions = env.transitions(s, action) # given a state, apply transitions for all of the actions
        # for each transition and its probabilities, update the sum for each action
        for s1, (reward, prob) in enumerate(transitions):
            sum1[action] += prob * (reward + gamma * V[s1])
    # before choosing the greedy option, we must clear s
    pi[s] = np.zeros(len(env.A))
    # as the greedy option it, the policy with the highest sum gets subset and assigned probability of 1
    pi[s][sum1.argmax()] = 1.00
    ### END CODE HERE ###
```

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

In [34]:
```
%reset_selective -f "^num_spaces$|^num_prices$|^env$|^V$|^pi$|^gamma$|^theta$"
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.

In [35]:
```
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

In [ ]:
```
print(pi)
```

```
[[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.]]
```

The cell below will check that your code passes the test case above. (Your code passed if the cell runs without error)

In [35]:
```
## Test Code ##
V_correct = [81.6,83.3,85.0,86.8,88.5,90.2,91.7,93.1,94.3,95.3,89.5]
pi_correct = [[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.]]
np.testing.assert_array_almost_equal(V, V_correct, 1)
print("correct value function")
np.testing.assert_array_almost_equal(pi, pi_correct)
print("Assertion correct. You can now move to the next section.")
```

```
correct value function
Assertion correct. You can now move to the next section.
```

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.

In [36]:
```
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
```

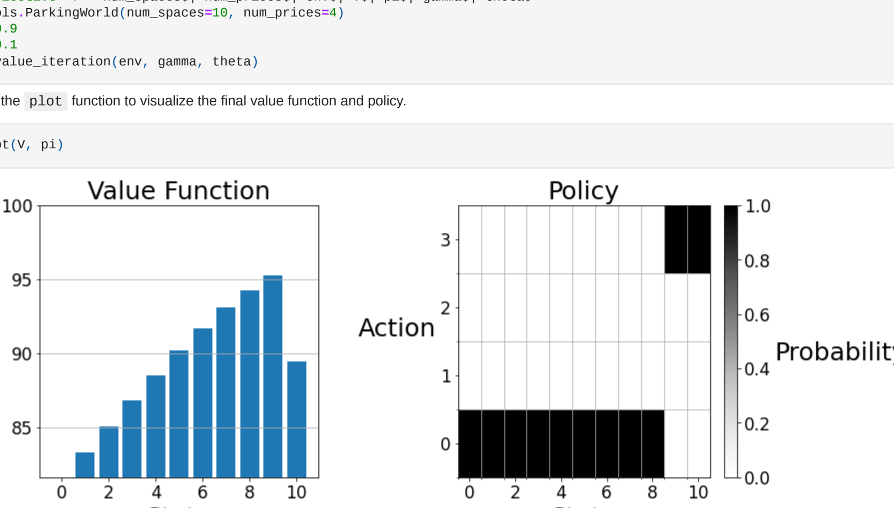
In [37]:
```
# [Graded]
def bellman_optimality_update(env, V, s, gamma):
    """
    Mutate ``V`` according to the Bellman optimality update equation.
    """
    ### START CODE HERE ###
    sum1 = np.zeros(len(env.A)) # number of actions in the environment initiated at zero
    for action in env.A:
      transitions = env.transitions(s, action) # given a state, apply transitions for all of the actions
      # for each transition and its probabilities, update the sum for each action
      for s1, (reward, prob) in enumerate(transitions):
            sum1[action] += prob * (reward + gamma * V[s1])
    V[s] = sum1[np.argmax(sum1)] # update V with the highest option

    ### END CODE HERE ###
```

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

In [38]:
```
%reset_selective -f "^num_spaces$|^num_prices$|^env$|^V$|^pi$|^gamma$|^theta$"
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.

In [39]:
```
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

The cell below will check that your code passes the test case above. (Your code passed if the cell runs without error)

In [40]:
```
## Test Code ##
V_correct = [81.6,83.3,85.0,86.8,88.5,90.2,91.7,93.1,94.3,95.3,89.5]
pi_correct = [[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.]]
np.testing.assert_array_almost_equal(V, V_correct, 1)
print("correct value function")
np.testing.assert_array_almost_equal(pi, pi_correct)
print("Assertion correct. You can now move to the next section.")
```

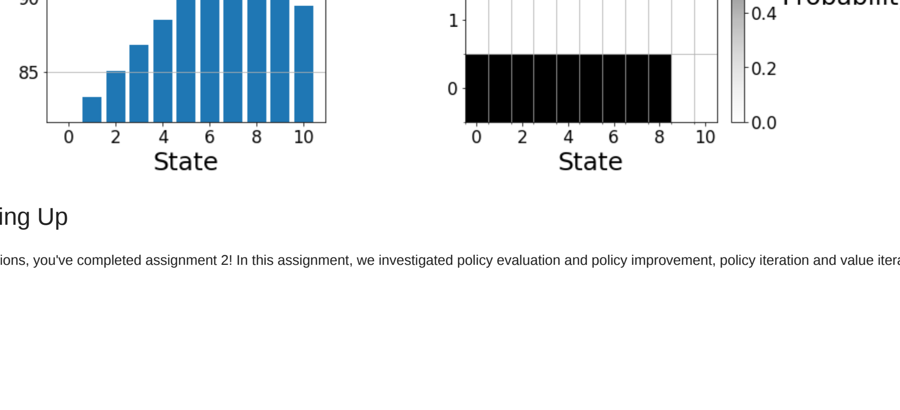
```
correct value function
Assertion correct. You can now move to the next section.
```

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.

In [41]:
```
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.

In [42]:
```
%reset_selective -f "^num_spaces$|^num_prices$|^env$|^V$|^pi$|^gamma$|^theta$"
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)
```



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!