Dynamic Programming

Welcome, This notebook will help you understand:

- Policy Evaluation and Policy Improvement.
- Value and Policy Iteration.
- Bellman Equations.

Georgian College Parking

Georgian College has a shortfall of parking spaces (next to department buildings) especially in winters. To address this, the college administration has decided to modify the pricing scheme to better promote social welfare. In general, the administration considers social welfare higher when more parking is being used, the exception being that the administration prefers that at least one spot is left unoccupied (so that it is available in case someone really needs it). The administration has created a Markov decision process (MDP) to model the demand for parking with a reward function that reflects its preferences. Now the Georgian College administration has hired you — an expert in dynamic programming — to help determine an optimal policy.

In [1]:
```
%%capture
%matplotlib inline
import numpy as np
import pickle
import tools
```

In [2]:
```
num_spaces = 3
num_prices = 3
env = tools.GCParking(num_spaces, num_prices)
V = np.zeros(num_spaces + 1)
pi = np.ones((num_spaces + 1, num_prices)) / num_prices
```

In [3]:
```
env
```

Out[3]:
```
<tools.GCParking at 0x11793a790>
```

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 [7]:
```
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
```

In [8]:
```
pi
```

Out[8]:
```
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]])
```

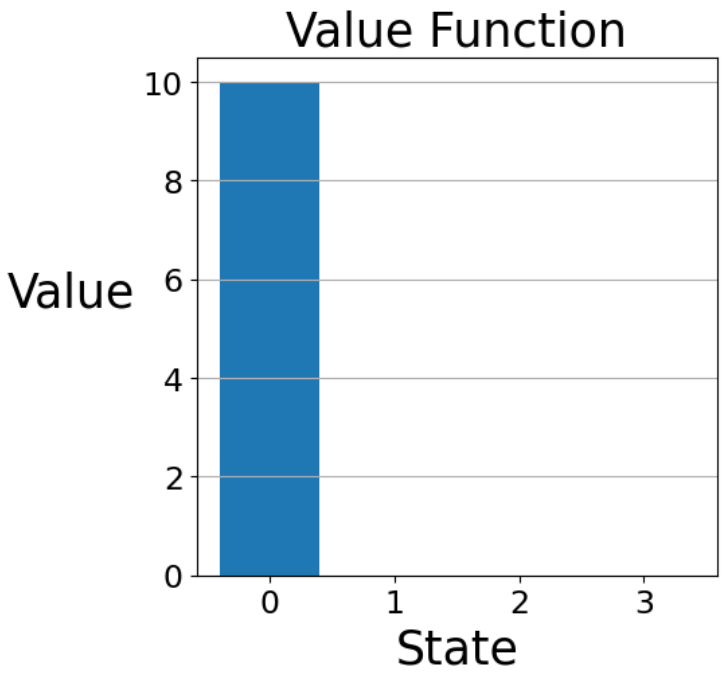
In [9]:
```
state = 0
pi[state]
```

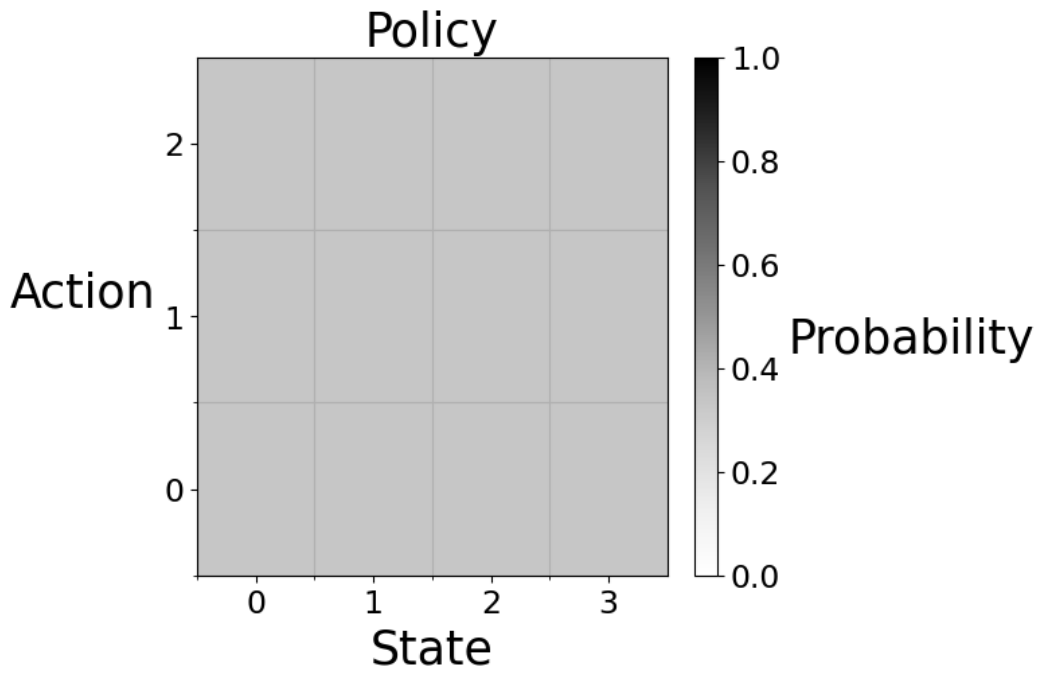
Out[9]:
```
array([0.33333333, 0.33333333, 0.33333333])
```

In [10]:
```
state = 0
action = 1
pi[state, action]
```

Out[10]:
```
0.3333333333333333
```

In [11]:
```
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 [12]:
```
env.S
```

Out[12]:
```
[0, 1, 2, 3]
```

In [13]:
```
env.A
```

Out[13]:
```
[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 [14]:
```
state = 3
action = 1
transitions = env.transitions(state, action)
transitions
```

Out[14]:
```
array([[1.        , 0.12390437],
       [2.        , 0.15133714],
       [3.        , 0.1848436 ],
       [2.        , 0.53991488]])
```

In [15]:
```
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 administration 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.

$$\large v(s) \leftarrow \sum_a \pi(a | s) \sum_{s', r} p(s', r | s, a)[r + \gamma v(s')]$$
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 [16]:
```
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
```

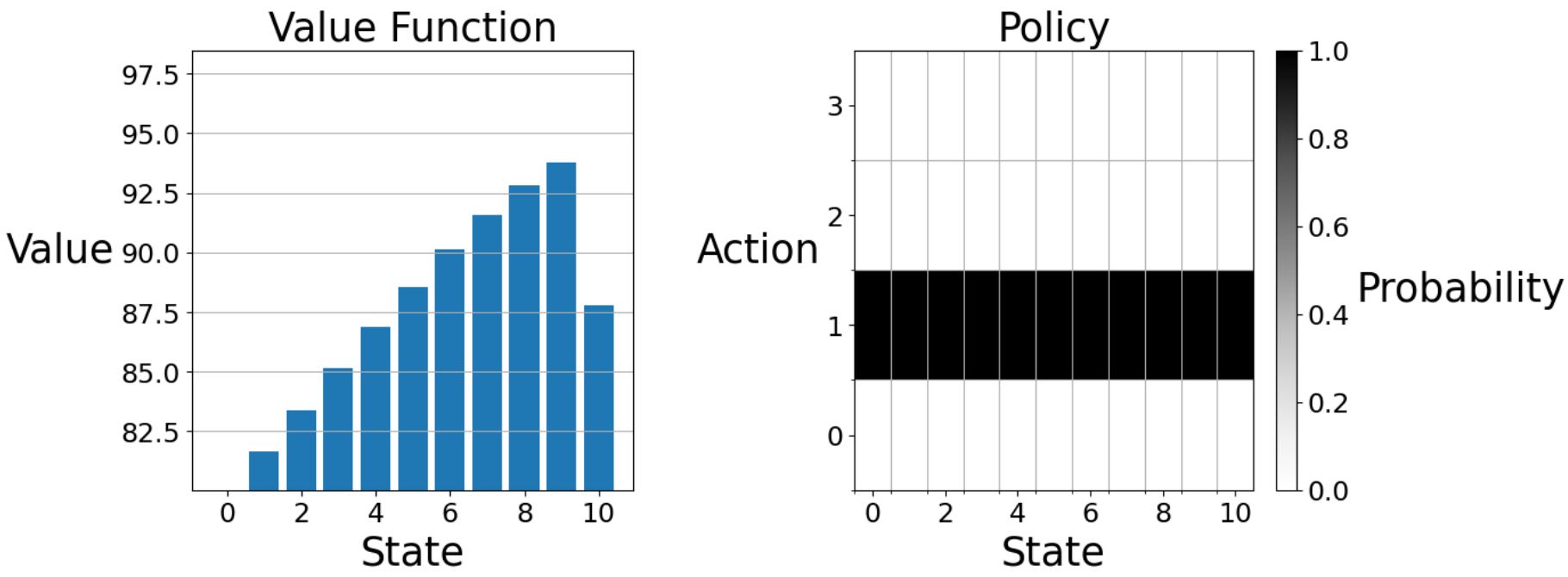
In [17]:
```
"""
Creating the function for named bellman_update to update the given value function "V" for given state "S" according

Here the function takes 5 main parameters in it (env, V, pi, s, gamma) :

1. "env" - is a variable which represent the environment, which provides us with set of actions and the transition
2. "V" - this variable stores the list of array for the current value function assigned to each state.
3. "pi" - here "pi" is a policy matrix which provides probability of taking action in state. ** pi[state, action] *
4. "s" - the variable "s" is stores the current state for which the value function is updated.
5. "gamma" - this is the discount factor, this is important for the future rewards.

"""

def bellman_update(env, V, pi, s, gamma):
    """Mutate ``V`` according to the Bellman update equation."""
    ### START CODE HERE ###

    a = 0
    for action in env.A:
        probability_of_action = pi[s, action]
        transitions = env.transitions(s, action)
        for new_state, (n_reward, n_probability) in enumerate(transitions):
            a += probability_of_action * n_probability * (n_reward + gamma * V[new_state])
    V[s] = a

    ### END CODE HERE ###

    """
    Step by step explanation for above code :

    1. Defined the variable "a" which will store the new value after updating it.
    2. Started a loop where it will iterate over all possible actions,
        - where variable "action" will represent the each action that can be taken in state "s" and
        - "env.A" here consists all possible set of actions available in our environment. In my understanding the "
    3. Storing the probability in variable "probability_of_action" :
        - which means that it will provide with the probability of taking action in state with respect to policy "p
        - where,
            * "s" - represnt the state and
            * "action" - represents action that can be taken in state s

    4. Getting the transation for action and state :
        - Here in this step I am fetching the information about what will happen if the agent performs a given acti
        - This will returns a list of possible transitions. Each transition will provide with the tuple containing
            - "reward" -  which the agent will receive after taking action "action" in state s.
            - "probability" - here in this step the probability means that getting probability of transitioning to


    5. Iterate over possible next states and their rewards and probabilities :
        - After defining the "transitions" from which we can get the "reward" and "probability" for new state. I in

    6. Updating the "a" (new valiue) by adding the expected value of taking action "action" in a state "s" with res
        - In terms of the Bellman equation this line Means that it computes the expected value of taking action "ac
        it then add this to new value here which is "a" which accumulates the total expected value for state "s" un

    7. At last update the value function for state "s" obtained from the above code


    """
```

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

In [18]:
```
%reset_selective -f "^num_spaces$|^num_prices$|^env$|^V$|^pi$|^gamma$|^theta$"
num_spaces = 10
num_prices = 4
env = tools.GCParking(num_spaces, num_prices)
V = np.zeros(num_spaces + 1)
gc_policy = np.zeros((num_spaces + 1, num_prices))
gc_policy[:, 1] = 1
gamma = 0.9
theta = 0.1
V = evaluate_policy(env, V, gc_policy, gamma, theta)
```

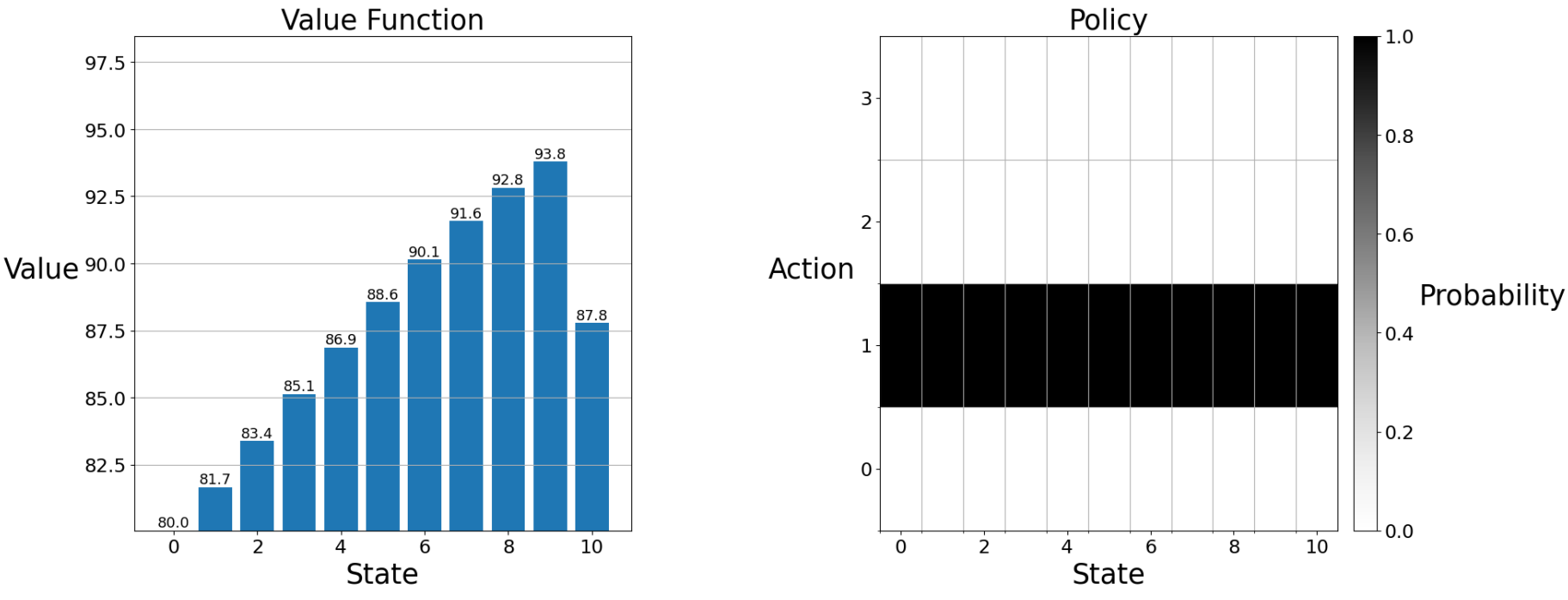
You can use the `plot` function to visualize the final value function and policy.

In [19]:
```
tools.plot(V, gc_policy)
```



Little modification done to the code present in tools.py file to plotting the graph with values to the each bar :

In [20]:
```
"""
Created the function name "plot_with_value" in tools.py  by modified the existing function "plot" in tools.py, to a
For better insight of the plot.
"""

tools.plot_with_values(V, gc_policy)
```



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

State $\quad\quad$ Value

0 $\quad\quad\quad\;$ 80.0

1 $\quad\quad\quad\;$ 81.7

2 $\quad\quad\quad\;$ 83.4

3 $\quad\quad\quad\;$ 85.1

4 $\quad\quad\quad\;$ 86.9

5 $\quad\quad\quad\;$ 88.6

6 $\quad\quad\quad\;$ 90.1

7 $\quad\quad\quad\;$ 91.6

8 $\quad\quad\quad\;$ 92.8

9 $\quad\quad\quad\;$ 93.8

10 $\quad\quad\;\;\,\,$ 87.8

Observe that the value function qualitatively resembles the administration'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).

Section 2: Policy Iteration

Now the administration 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 [21]:
```
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
```

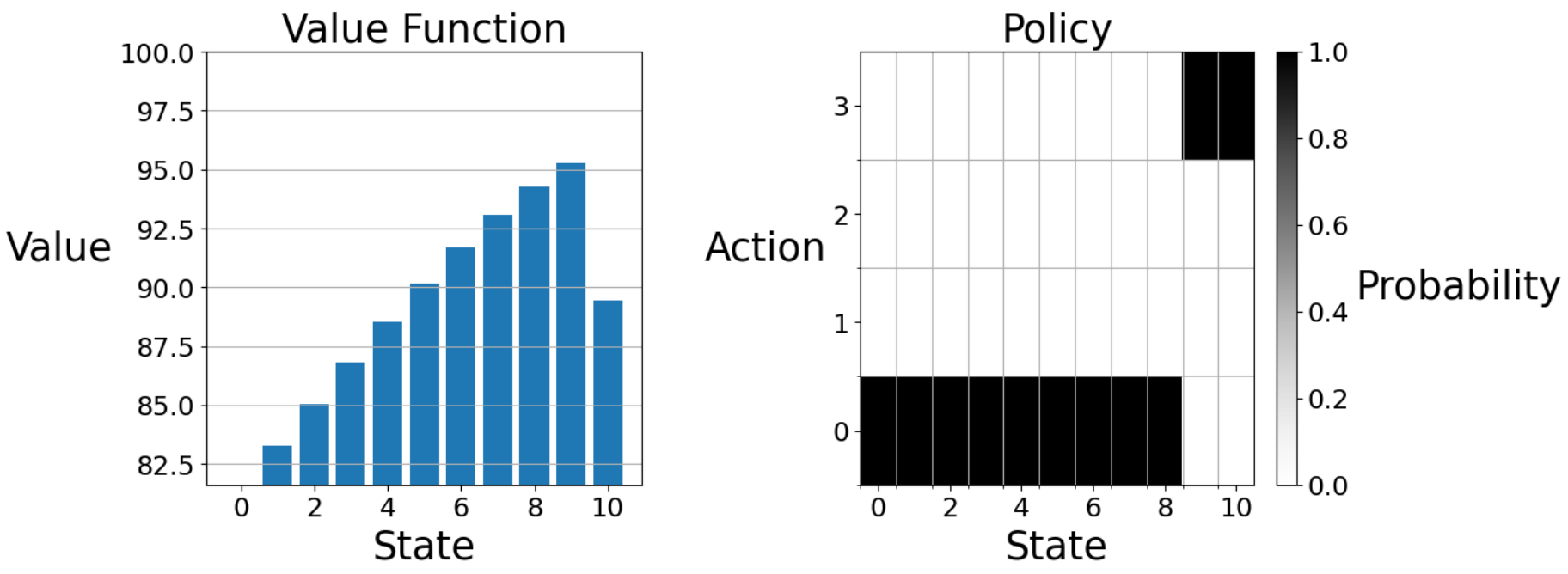
In [22]:
```
# [Graded]
def q_greedify_policy(env, V, pi, s, gamma):
    """Mutate ``pi`` to be greedy with respect to the Q-values induced by ``V``."""
    Q = np.zeros(len(env.A))

    for action in env.A:
        q = 0
        transitions = env.transitions(s, action)
        for s_prime, (r, p) in enumerate(transitions):
            q += p * (r + gamma * V[s_prime])
        Q[action] = q

    best_action = np.argmax(Q)

    for action in env.A:
        if action == best_action:
            pi[s, action] = 1.0
        else:
            pi[s, action] = 0.0
    ### END CODE HERE ###
```

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

In [23]:
```
%reset_selective -f "^num_spaces$|^num_prices$|^env$|^V$|^pi$|^gamma$|^theta$"
env = tools.GCParking(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 [24]:
```
tools.plot(V, pi)
```



You can check the value function (rounded to one decimal place) and policy against the answer below:
State $\quad\quad$ Value $\quad\quad$ Action

0 $\quad\quad\quad\;$ 81.6 $\quad\quad\;$ 0
1 $\quad\quad\quad\;$ 83.3 $\quad\quad\;$ 0
2 $\quad\quad\quad\;$ 85.0 $\quad\quad\;$ 0
3 $\quad\quad\quad\;$ 86.8 $\quad\quad\;$ 0
4 $\quad\quad\quad\;$ 88.5 $\quad\quad\;$ 0
5 $\quad\quad\quad\;$ 90.2 $\quad\quad\;$ 0
6 $\quad\quad\quad\;$ 91.7 $\quad\quad\;$ 0
7 $\quad\quad\quad\;$ 93.1 $\quad\quad\;$ 0
8 $\quad\quad\quad\;$ 94.3 $\quad\quad\;$ 0
9 $\quad\quad\quad\;$ 95.3 $\quad\quad\;$ 3
10 $\quad\quad\;\;\;\,$ 89.5 $\quad\quad\;$ 3

Section 3: Value Iteration

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

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

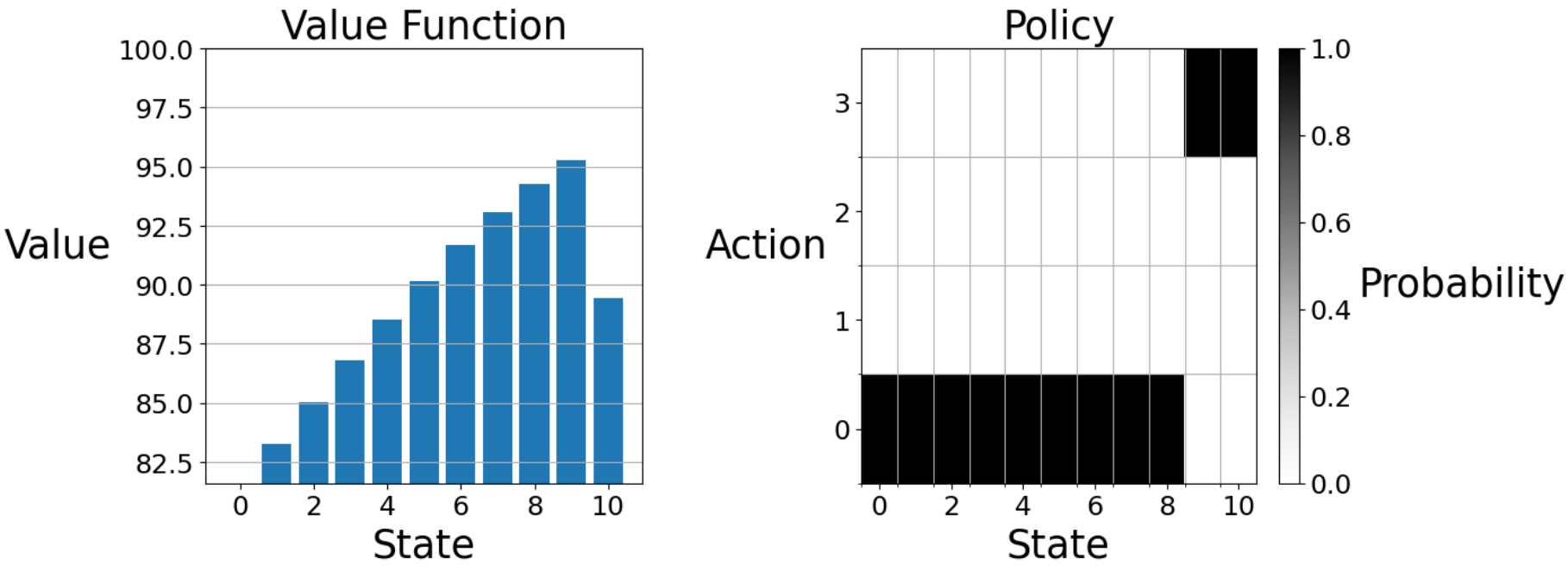
In [25]:
```
def value_iteration(env, gamma, theta):
    V = np.zeros(len(env.S))
    while True:
        delta = 0
        for s in env.S:
            v = V[s]
            bellman_optimality_update(env, V, s, gamma)
            delta = max(delta, abs(v - V[s]))
        if delta < theta:
            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 [31]:
```
# [Graded]
def bellman_optimality_update(env, V, s, gamma):
    """Mutate ` V ` according to the Bellman optimality update equation."""

    # Initialize G to a list of zeros by multiplying 0s to len of actions
    G = [0] * len(env.A)

    for action_index, action in enumerate(env.A):

        #Fetching the transition of state s and action.
        transitions = env.transitions(s, action)

        # Iterate over each transition.
        for s_index, (r, p) in enumerate(transitions):

            # Calculating disconted reward with gamma-decay
            discounted_reward = gamma * V[s_index]

            # Value function
            G[action_index] += p * ( r + discounted_reward)

    V[s] = max(G)
```

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

In [27]:
```
%reset_selective -f "^num_spaces$|^num_prices$|^env$|^V$|^pi$|^gamma$|^theta$"
env = tools.GCParking(num_spaces=10, num_prices=4)
gamma = 0.9
theta = 0.1
V, pi = value_iteration(env, gamma, theta)
```

In [28]:
```
tools.plot(V, pi)
```



You can check your value function (rounded to one decimal place) and policy against the answer below:

State $\quad\quad$ Value $\quad\quad$ Action

0 $\quad\quad\quad\;$ 81.6 $\quad\quad\;$ 0

1 $\quad\quad\quad\;$ 83.3 $\quad\quad\;$ 0

2 $\quad\quad\quad\;$ 85.0 $\quad\quad\;$ 0

3 $\quad\quad\quad\;$ 86.8 $\quad\quad\;$ 0

4 $\quad\quad\quad\;$ 88.5 $\quad\quad\;$ 0

5 $\quad\quad\quad\;$ 90.2 $\quad\quad\;$ 0

6 $\quad\quad\quad\;$ 91.7 $\quad\quad\;$ 0

7 $\quad\quad\quad\;$ 93.1 $\quad\quad\;$ 0

8 $\quad\quad\quad\;$ 94.3 $\quad\quad\;$ 0

9 $\quad\quad\quad\;$ 95.3 $\quad\quad\;$ 3

10 $\quad\quad\;\;\,$ 89.5 $\quad\quad\;$ 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.

In [29]:
```
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
```

In [30]:
```
%reset_selective -f "^num_spaces$|^num_prices$|^env$|^V$|^pi$|^gamma$|^theta$"
env = tools.GCParking(num_spaces=10, num_prices=4)
gamma = 0.9
theta = 0.1
V, pi = value_iteration2(env, gamma, theta)
tools.plot(V, pi)
```