ce-game-using-dynammic-programming

July 9, 2024

0.1 Dice game Dynammic Programming

0.1.1 Group ID: DRL GROUP 73

0.1.2 Group Members Name with Student ID:

1.Gujja Anirudh - 2023AA05236 2.GUNTUKU ANANDA RAJ - 2023AB05189 3.S SUJANA - 2023AB05187 4.RATNABABU MAMIDI - 2023AB05147

[37]:
```python
import numpy as np

# Constants
goal = 100
gamma = 1.0
prob_roll = 1/6

# Initialize value function and policy
V = np.zeros(goal + 1)
policy = np.zeros(goal + 1, dtype=int)  # 0 for "stop", 1 for "roll"
```

[38]:
```python
##Dice Enviironment
class DiceGameEnvironment:
    def __init__(self, goal):
        """
        Initializes the Dice Game environment with a specified goal.

        Parameters:
        - goal (int): The target goal to achieve in the game.
        """
        self.goal = goal

    def step(self, state, action):
        """
        Executes a step (action) in the Dice Game environment.

        Parameters:
        - state (int): The current state or points accumulated.
        - action (str): The action to take. Can be "stop" or "roll".
```

```
        Returns:
        - tuple: (new_state, reward)
            - new_state (int): The new state after taking the action.
            - reward (int): The reward obtained from the action.
        """
        if action == "stop":
            # If action is "stop", return current state and reward based on␣
↪goal achievement
            return state, state if state == self.goal else 0
        elif action == "roll":
            # If action is "roll", roll a dice and update state accordingly
            roll = np.random.randint(1, 7)  # Roll a dice (1 to 6)
            if roll == 1:
                return 0, 0  # If rolled 1, reset state to 0 and reward to 0
            else:
                new_state = state + roll  # Otherwise, update state with the␣
↪roll
                return new_state, new_state if new_state == self.goal else 0  #␣
↪Check if goal is achieved
```

[39]:
```
##Reward Function
def calculate_reward(state, action):
    """
    Calculates the expected reward for a given state and action in a dice game␣
↪environment.

    Parameters:
    - state (int): The current state or points accumulated in the game.
    - action (str): The action to take. Can be "stop" or "roll".

    Returns:
    - float: The expected reward for the given action and state.
    """
    if action == "stop":
        # If action is "stop", return current state if it matches goal;␣
↪otherwise, return 0
        return state if state == goal else 0
    elif action == "roll":
        rewards = [0] * 6  # Initialize rewards list for each possible dice␣
↪roll (1 to 6)
        for roll in range(1, 7):
            if roll == 1:
                rewards[roll - 1] = 0  # If rolled 1, reward is 0
            else:
                new_state = state + roll  # Calculate new state after rolling␣
↪the dice
```

```
                rewards[roll - 1] = new_state if new_state == goal else 0  # Reward is new state if goal is achieved; otherwise, 0

        return np.mean(rewards)  # Return the mean of rewards as the expected reward
```

[40]:
```
##Policy iteration
def policy_evaluation(policy, V, gamma=1.0, theta=1e-4):
    """
    Evaluates the value function V under a given policy.

    Parameters:
    - policy (list): The policy mapping for each state.
    - V (list): The value function for each state.
    - gamma (float, optional): Discount factor for future rewards (default: 1.0).
    - theta (float, optional): Threshold for stopping iteration (default: 1e-4).

    Returns:
    - list: The updated value function V.
    """
    while True:
        delta = 0
        for state in range(1, goal + 1):  # Loop through all states from 1 to goal
            v = V[state]
            if policy[state] == 0:  # stop action
                V[state] = calculate_reward(state, "stop")
            else:  # roll action
                # Calculate expected rewards for rolling
                roll_rewards = [prob_roll * V[state + roll] if state + roll <= goal else 0 for roll in range(2, 7)]
                V[state] = np.mean(roll_rewards)
            delta = max(delta, abs(v - V[state]))  # Calculate maximum change in value function
        if delta < theta:  # Check if convergence criteria met
            break
    return V

def policy_improvement(policy, V, gamma=1.0):
    """
    Improves the policy based on the current value function V.

    Parameters:
    - policy (list): The policy mapping for each state.
    - V (list): The value function for each state.
```

```
    - gamma (float, optional): Discount factor for future rewards (default: 1.
 ↪0).

    Returns:
    - tuple: Updated policy and a boolean indicating if the policy is stable.
    """
    stable = True
    for state in range(1, goal + 1):  # Loop through all states from 1 to goal
        old_action = policy[state]

        # Calculate rewards for stopping and rolling actions
        stop_reward = calculate_reward(state, "stop")
        roll_reward = prob_roll * (sum([V[state + roll] if state + roll <= goal␣
 ↪else 0 for roll in range(2, 7)]) / 5)

        if stop_reward >= roll_reward:
            policy[state] = 0  # Set policy to stop action
        else:
            policy[state] = 1  # Set policy to roll action

        if old_action != policy[state]:
            stable = False  # Policy not stable if action changes

    return policy, stable

def policy_iteration(policy, V, gamma=1.0):
    """
    Performs policy iteration to find an optimal policy and value function.

    Parameters:
    - policy (list): The initial policy mapping for each state.
    - V (list): The initial value function for each state.
    - gamma (float, optional): Discount factor for future rewards (default: 1.
 ↪0).

    Returns:
    - tuple: Optimal policy and value function.
    """
    while True:
        V = policy_evaluation(policy, V, gamma)  # Evaluate current policy
        policy, stable = policy_improvement(policy, V, gamma)  # Improve policy␣
 ↪based on evaluation
        if stable:
            break  # Break loop if policy is stable (no changes in policy)
    return policy, V
```

[41]:
```
#Value iteration function
def value_iteration(V, gamma=1.0, theta=1e-4):
    """
    Performs value iteration to find the optimal value function V*.

    Parameters:
    - V (list): The initial value function for each state.
    - gamma (float, optional): Discount factor for future rewards (default: 1.
 ↪0).
    - theta (float, optional): Threshold for stopping iteration (default: 1e-4).

    Returns:
    - list: The optimal value function V*.
    """
    while True:
        delta = 0
        for state in range(1, goal + 1):  # Loop through all states from 1 to␣
 ↪goal
            v = V[state]

            # Calculate expected rewards for rolling
            roll_rewards = [prob_roll * V[state + roll] if state + roll <= goal␣
 ↪else 0 for roll in range(2, 7)]

            # Update value function using Bellman optimality equation
            V[state] = max(calculate_reward(state, "stop"), np.
 ↪mean(roll_rewards))

            delta = max(delta, abs(v - V[state]))  # Calculate maximum change␣
 ↪in value function

        if delta < theta:  # Check if convergence criteria met
            break

    return V
```

[42]:
```
##Executing Policy Iteration
env = DiceGameEnvironment(goal)
optimal_policy, optimal_value = policy_iteration(policy, V, gamma)

# Print results
print("Policy Iteration - Optimal Policy:")
print(optimal_policy)
```

```
Policy Iteration - Optimal Policy:
[0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

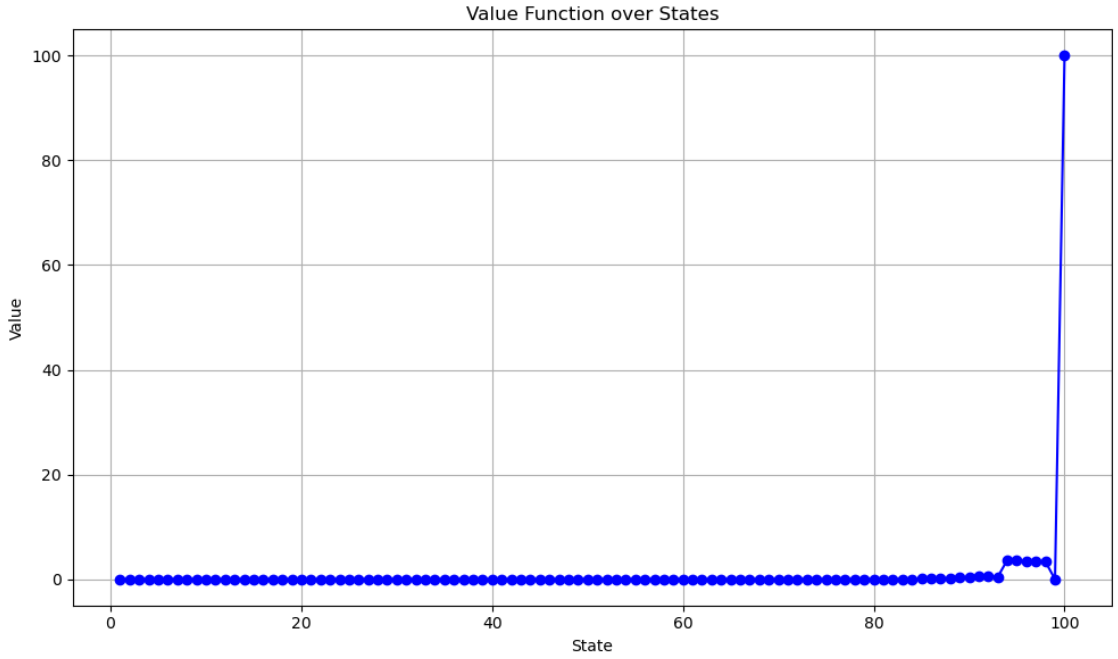
```
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0]
```

[43]:
```
#Executing Value Iteration
env = DiceGameEnvironment(goal)
optimal_value = value_iteration(V, gamma)

# Print results
print("Value Iteration - Optimal Value Function:")
print(optimal_value)
```

```
Value Iteration - Optimal Value Function:
[0.00000000e+00 4.27774519e-18 1.00258335e-17 2.21433113e-17
 4.63901700e-17 9.27051528e-17 1.77587435e-16 3.27545690e-16
 5.84047340e-16 1.01062928e-15 1.70320553e-15 2.80519869e-15
 4.53003225e-15 7.19489189e-15 1.12717172e-14 1.74644026e-14
 2.68256013e-14 4.09330801e-14 6.21554674e-14 9.40522769e-14
 1.41976155e-13 2.13978948e-13 3.22172710e-13 4.84772234e-13
 7.29162677e-13 1.09651425e-12 1.64874103e-12 2.47896719e-12
 3.72721519e-12 5.60392220e-12 8.42531707e-12 1.26668018e-11
 1.90437536e-11 2.86324441e-11 4.30502200e-11 6.47257303e-11
 9.73081188e-11 1.46287963e-10 2.19940799e-10 3.30710824e-10
 4.97258947e-10 7.47573397e-10 1.12375961e-09 1.68933613e-09
 2.54029588e-09 3.82035972e-09 5.74401708e-09 8.63319310e-09
 1.29749224e-08 1.95075915e-08 2.93491522e-08 4.41459322e-08
 6.63429139e-08 9.96502031e-08 1.49759472e-07 2.25329225e-07
 3.39392753e-07 5.10246314e-07 7.65559654e-07 1.14897815e-06
 1.72860729e-06 2.60648535e-06 3.93215214e-06 5.89116649e-06
 8.80837836e-06 1.32311621e-05 1.99953595e-05 3.02684942e-05
 4.56611702e-05 6.75788086e-05 1.00747518e-04 1.52678872e-04
 2.33194416e-04 3.53855212e-04 5.29359090e-04 7.58276669e-04
 1.14774016e-03 1.79113502e-03 2.76932153e-03 4.14918298e-03
 6.02339300e-03 8.01526754e-03 1.34750398e-02 2.20711672e-02
 3.34947784e-02 4.74192364e-02 6.42415684e-02 7.32312757e-02
 1.85864335e-01 2.91378601e-01 3.90127572e-01 4.81975309e-01
 5.77901235e-01 4.55555556e-01 3.67037037e+00 3.55555556e+00
 3.44444444e+00 3.33333333e+00 3.33333333e+00 0.00000000e+00
 1.00000000e+02]
```

[44]:
```
# Line plot to visualize the value function over different states
plt.figure(figsize=(10, 6))
states = np.arange(1, goal + 1)
plt.plot(states, optimal_value[1:], marker='o', linestyle='-', color='b')
plt.title('Value Function over States')
plt.xlabel('State')
plt.ylabel('Value')
plt.grid(True)
plt.tight_layout()
```

```
plt.show()
```



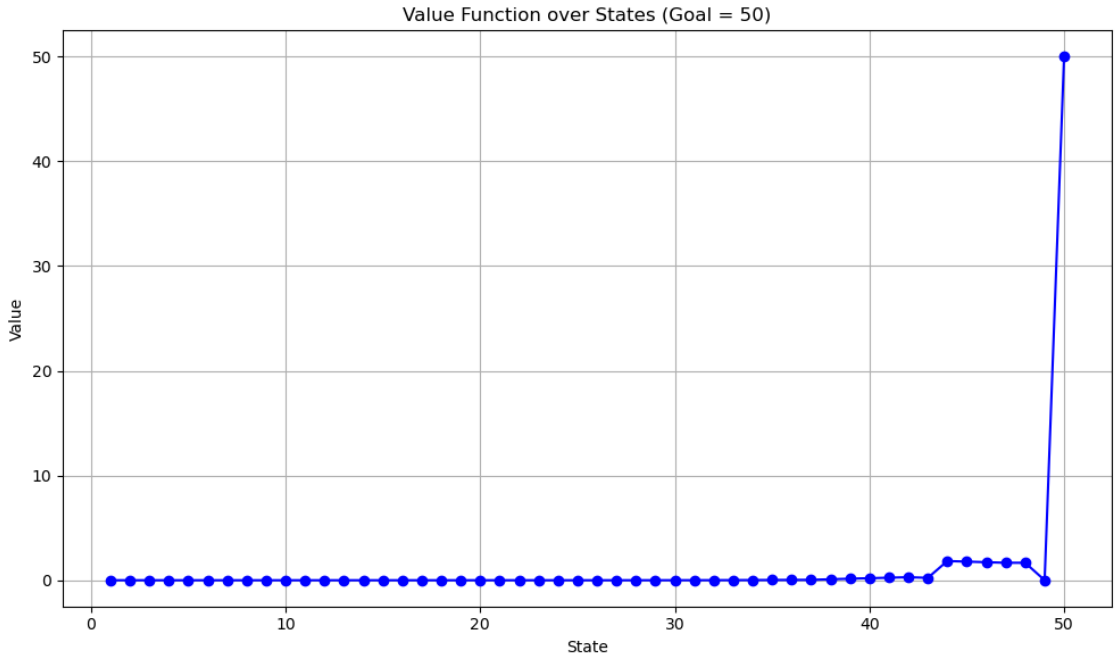
[45]:
```
##when goal score is 50
import numpy as np
import matplotlib.pyplot as plt

# Constants
goal = 50  # Change goal to 50
gamma = 1.0
prob_roll = 1/6  # Probability of not rolling a 1

# Initialize value function and policy
V = np.zeros(goal + 1)
policy = np.zeros(goal + 1, dtype=int)  # 0 for "stop", 1 for "roll"

class DiceGameEnvironment:
    def __init__(self, goal):
        self.goal = goal

    def step(self, state, action):
        if action == "stop":
            return state, state if state == self.goal else 0
        elif action == "roll":
            roll = np.random.randint(1, 7)
            if roll == 1:
```

```
                return 0, 0
            else:
                new_state = state + roll
                return new_state, new_state if new_state == self.goal else 0

def calculate_reward(state, action):
    if action == "stop":
        return state if state == goal else 0
    elif action == "roll":
        rewards = [0] * 6
        for roll in range(1, 7):
            if roll == 1:
                rewards[roll - 1] = 0
            else:
                new_state = state + roll
                rewards[roll - 1] = new_state if new_state == goal else 0
        return np.mean(rewards)

def policy_evaluation(policy, V, gamma=1.0, theta=1e-4):
    while True:
        delta = 0
        for state in range(1, goal + 1):
            v = V[state]
            if policy[state] == 0:  # stop
                V[state] = calculate_reward(state, "stop")
            else:  # roll
                roll_rewards = [prob_roll * V[state + roll] if state + roll <=␣
 ↪goal else 0 for roll in range(2, 7)]
                V[state] = np.mean(roll_rewards)
            delta = max(delta, abs(v - V[state]))
        if delta < theta:
            break
    return V

def policy_improvement(policy, V, gamma=1.0):
    stable = True
    for state in range(1, goal + 1):
        old_action = policy[state]

        stop_reward = calculate_reward(state, "stop")
        roll_reward = prob_roll * (sum([V[state + roll] if state + roll <= goal␣
 ↪else 0 for roll in range(2, 7)]) / 5)

        if stop_reward >= roll_reward:
            policy[state] = 0  # stop
        else:
            policy[state] = 1  # roll
```

```
        if old_action != policy[state]:
            stable = False

    return policy, stable

def policy_iteration(policy, V, gamma=1.0):
    while True:
        V = policy_evaluation(policy, V, gamma)
        policy, stable = policy_improvement(policy, V, gamma)
        if stable:
            break
    return policy, V

def value_iteration(V, gamma=1.0, theta=1e-4):
    while True:
        delta = 0
        for state in range(1, goal + 1):
            v = V[state]
            roll_rewards = [prob_roll * V[state + roll] if state + roll <= goal␣
 ↪else 0 for roll in range(2, 7)]
            V[state] = max(calculate_reward(state, "stop"), np.
 ↪mean(roll_rewards))
            delta = max(delta, abs(v - V[state]))
        if delta < theta:
            break
    return V

# Initialize environment and run policy iteration for goal = 50
env = DiceGameEnvironment(goal)
optimal_policy, optimal_value = policy_iteration(policy, V, gamma)

# Print results
print("Policy Iteration - Optimal Policy for goal = 50:")
print(optimal_policy)
print("Policy Iteration - Optimal Value Function for goal = 50:")
print(optimal_value)

# Line plot to visualize the value function over different states
plt.figure(figsize=(10, 6))
states = np.arange(1, goal + 1)
plt.plot(states, optimal_value[1:], marker='o', linestyle='-', color='b')
plt.title('Value Function over States (Goal = 50)')
plt.xlabel('State')
plt.ylabel('Value')
plt.grid(True)
plt.tight_layout()
```

```
plt.show()
```

```
Policy Iteration - Optimal Policy for goal = 50:
[0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 0 0]
Policy Iteration - Optimal Value Function for goal = 50:
[0.00000000e+00 1.32796840e-08 2.42623997e-08 4.15221605e-08
 6.78267144e-08 1.07197974e-07 1.65868427e-07 2.52746688e-07
 3.81500425e-07 5.73894225e-07 8.64084161e-07 1.30319015e-06
 1.96607259e-06 2.94558272e-06 4.40418918e-06 6.61558105e-06
 9.99767974e-06 1.51342471e-05 2.28305851e-05 3.37894043e-05
 5.03737591e-05 7.63394358e-05 1.16597208e-04 1.76927606e-04
 2.64679545e-04 3.79138334e-04 5.73870081e-04 8.95567508e-04
 1.38466077e-03 2.07459149e-03 3.01169650e-03 4.00763377e-03
 6.73751989e-03 1.10355836e-02 1.67473892e-02 2.37096182e-02
 3.21207842e-02 3.66156379e-02 9.29321674e-02 1.45689300e-01
 1.95063786e-01 2.40987654e-01 2.88950617e-01 2.27777778e-01
 1.83518519e+00 1.77777778e+00 1.72222222e+00 1.66666667e+00
 1.66666667e+00 0.00000000e+00 5.00000000e+01]
```



##Observation Reducing the goal score from 100 to 50 leads to a more conservative optimal policy that favors stopping earlier and higher optimal values near the goal state, reflecting the reduced risk and increased reward associated with achieving the lower goal.

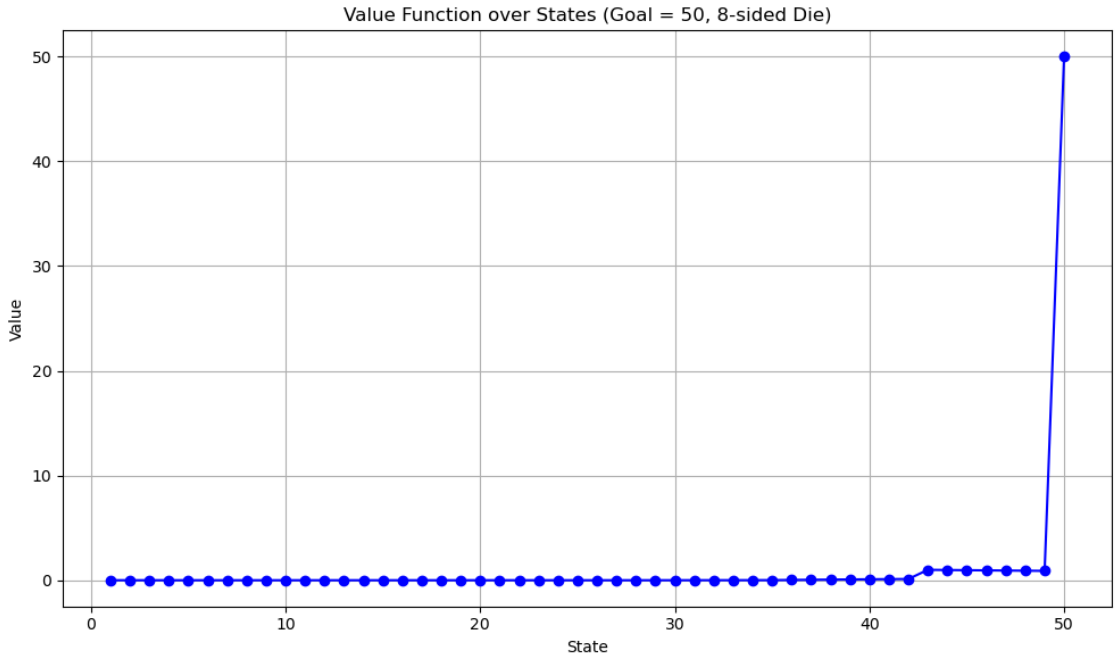
###How would the optimal policy and value function change if the die had 8 sides instead of 6? Assume the outcomes range from 0 to 7, with each outcome having a probability of 1/8.

[48]:
```
import numpy as np
import matplotlib.pyplot as plt

# Constants
goal = 50  # Change goal to 50
gamma = 1.0
prob_roll = 1/8  # Probability of not rolling a 0

# Initialize value function and policy
V = np.zeros(goal + 1)
policy = np.zeros(goal + 1, dtype=int)  # 0 for "stop", 1 for "roll"

class DiceGameEnvironment:
    def __init__(self, goal):
        self.goal = goal

    def step(self, state, action):
        if action == "stop":
            return state, state if state == self.goal else 0
        elif action == "roll":
            roll = np.random.randint(0, 8)  # 8-sided die roll (0 to 7)
            if roll == 0:
                return 0, 0
            else:
                new_state = state + roll
                return new_state, new_state if new_state == self.goal else 0

def calculate_reward(state, action):
    if action == "stop":
        return state if state == goal else 0
    elif action == "roll":
        rewards = [0] * 8
        for roll in range(1, 8):
            new_state = state + roll
            rewards[roll] = new_state if new_state == goal else 0
        return np.mean(rewards)

def policy_evaluation(policy, V, gamma=1.0, theta=1e-4):
    while True:
        delta = 0
        for state in range(1, goal + 1):
            v = V[state]
            if policy[state] == 0:  # stop
                V[state] = calculate_reward(state, "stop")
            else:  # roll
                roll_rewards = [prob_roll * V[state + roll] if state + roll <=␣
 ↪goal else 0 for roll in range(1, 8)]
```

```
                V[state] = np.mean(roll_rewards)
            delta = max(delta, abs(v - V[state]))
        if delta < theta:
            break
    return V

def policy_improvement(policy, V, gamma=1.0):
    stable = True
    for state in range(1, goal + 1):
        old_action = policy[state]

        stop_reward = calculate_reward(state, "stop")
        roll_reward = prob_roll * (sum([V[state + roll] if state + roll <= goal
 ↪else 0 for roll in range(1, 8)]) / 7)

        if stop_reward >= roll_reward:
            policy[state] = 0  # stop
        else:
            policy[state] = 1  # roll

        if old_action != policy[state]:
            stable = False

    return policy, stable

def policy_iteration(policy, V, gamma=1.0):
    while True:
        V = policy_evaluation(policy, V, gamma)
        policy, stable = policy_improvement(policy, V, gamma)
        if stable:
            break
    return policy, V

def value_iteration(V, gamma=1.0, theta=1e-4):
    while True:
        delta = 0
        for state in range(1, goal + 1):
            v = V[state]
            roll_rewards = [prob_roll * V[state + roll] if state + roll <= goal
 ↪else 0 for roll in range(1, 8)]
            V[state] = max(calculate_reward(state, "stop"), np.
 ↪mean(roll_rewards))
            delta = max(delta, abs(v - V[state]))
        if delta < theta:
            break
    return V
```

```python
# Initialize environment and run policy iteration for goal = 50
env = DiceGameEnvironment(goal)
optimal_policy, optimal_value = policy_iteration(policy, V, gamma)

# Print results
print("Policy Iteration - Optimal Policy for goal = 50 (8-sided die):")
print(optimal_policy)
print("Policy Iteration - Optimal Value Function for goal = 50 (8-sided die):")
print(optimal_value)

# Line plot to visualize the value function over different states
plt.figure(figsize=(10, 6))
states = np.arange(1, goal + 1)
plt.plot(states, optimal_value[1:], marker='o', linestyle='-', color='b')
plt.title('Value Function over States (Goal = 50, 8-sided Die)')
plt.xlabel('State')
plt.ylabel('Value')
plt.grid(True)
plt.tight_layout()
plt.show()
```

```
Policy Iteration - Optimal Policy for goal = 50 (8-sided die):
[0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 0]
Policy Iteration - Optimal Value Function for goal = 50 (8-sided die):
[0.00000000e+00 2.60991196e-10 1.21740712e-09 3.57450281e-09
 8.42926349e-09 1.74129537e-08 3.28534946e-08 5.79559320e-08
 9.77281281e-08 1.60861322e-07 2.60968350e-07 4.17666815e-07
 6.57515892e-07 1.01481512e-06 1.53227321e-06 2.34389566e-06
 3.66640408e-06 5.79156443e-06 9.07892822e-06 1.39489702e-05
 2.08766062e-05 3.03850749e-05 4.76934823e-05 7.76497622e-05
 1.24747029e-04 1.93139354e-04 2.86656988e-04 4.08821040e-04
 5.62857637e-04 1.01696353e-03 1.75520117e-03 2.76219389e-03
 4.02310956e-03 5.52364450e-03 7.25000798e-03 9.18890703e-03
 2.64468936e-02 4.30965086e-02 5.91537866e-02 7.46343868e-02
 8.95536012e-02 1.03926363e-01 1.17767254e-01 9.92894144e-01
 9.75474949e-01 9.58361353e-01 9.41547996e-01 9.25029610e-01
 9.08801020e-01 8.92857143e-01 5.00000000e+01]
```



###Experiment with different discount factors (e.g., 0.9, 0.95). How does discounting future rewards impact the optimal policy and value function?

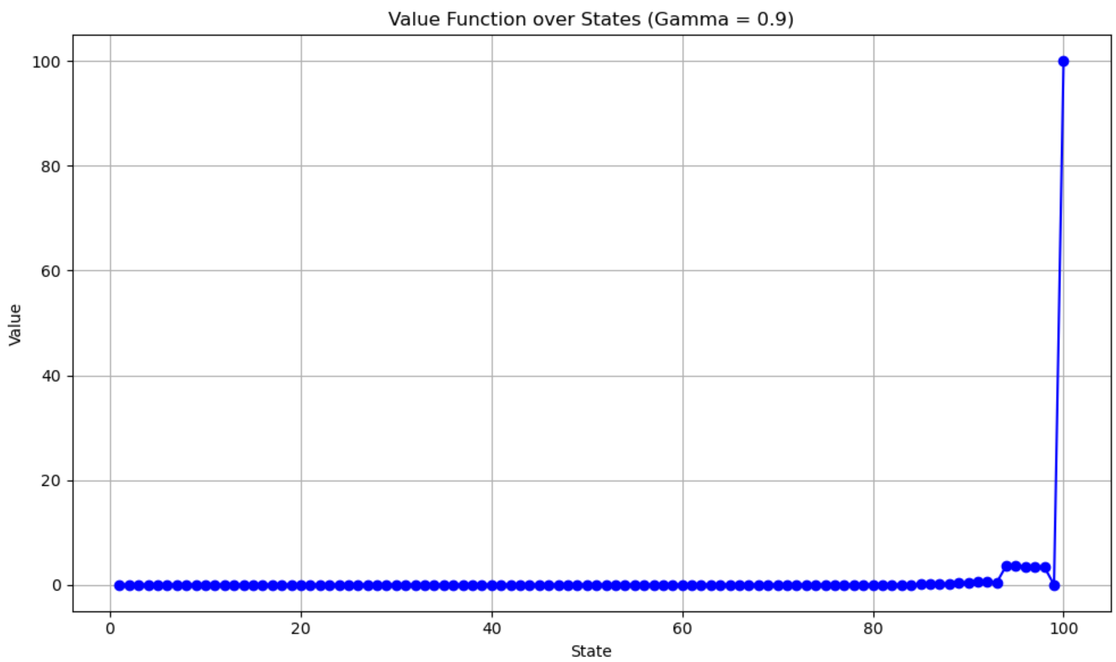
[50]:
```python
import numpy as np
import matplotlib.pyplot as plt

# Constants
goal = 100
discount_factors = [0.9, 0.95]
prob_roll = 1/6  # Probability of not rolling a 1

class DiceGameEnvironment:
    def __init__(self, goal):
        self.goal = goal

    def step(self, state, action):
        if action == "stop":
            return state, state if state == self.goal else 0
        elif action == "roll":
            roll = np.random.randint(1, 7)
            if roll == 1:
                return 0, 0
            else:
                new_state = state + roll
                return new_state, new_state if new_state == self.goal else 0
```

```python
def calculate_reward(state, action):
    if action == "stop":
        return state if state == goal else 0
    elif action == "roll":
        rewards = [0] * 6
        for roll in range(1, 7):
            if roll == 1:
                rewards[roll - 1] = 0
            else:
                new_state = state + roll
                rewards[roll - 1] = new_state if new_state == goal else 0
        return np.mean(rewards)

def policy_evaluation(policy, V, gamma=1.0, theta=1e-4):
    while True:
        delta = 0
        for state in range(1, goal + 1):
            v = V[state]
            if policy[state] == 0:  # stop
                V[state] = calculate_reward(state, "stop")
            else:  # roll
                roll_rewards = [prob_roll * V[state + roll] if state + roll <=␣
 ↪goal else 0 for roll in range(2, 7)]
                V[state] = np.mean(roll_rewards)
            delta = max(delta, abs(v - V[state]))
        if delta < theta:
            break
    return V

def policy_improvement(policy, V, gamma=1.0):
    stable = True
    for state in range(1, goal + 1):
        old_action = policy[state]

        stop_reward = calculate_reward(state, "stop")
        roll_reward = prob_roll * (sum([V[state + roll] if state + roll <= goal␣
 ↪else 0 for roll in range(2, 7)]) / 5)

        if stop_reward >= roll_reward:
            policy[state] = 0  # stop
        else:
            policy[state] = 1  # roll

        if old_action != policy[state]:
            stable = False

    return policy, stable
```

```
def policy_iteration(policy, V, gamma=1.0):
    while True:
        V = policy_evaluation(policy, V, gamma)
        policy, stable = policy_improvement(policy, V, gamma)
        if stable:
            break
    return policy, V

def value_iteration(V, gamma=1.0, theta=1e-4):
    while True:
        delta = 0
        for state in range(1, goal + 1):
            v = V[state]
            roll_rewards = [prob_roll * V[state + roll] if state + roll <= goal␣
 ↪else 0 for roll in range(2, 7)]
            V[state] = max(calculate_reward(state, "stop"), np.
 ↪mean(roll_rewards))
            delta = max(delta, abs(v - V[state]))
        if delta < theta:
            break
    return V

# Experimenting with different discount factors
for gamma in discount_factors:
    V = np.zeros(goal + 1)
    policy = np.zeros(goal + 1, dtype=int)

    # Run policy iteration
    env = DiceGameEnvironment(goal)
    optimal_policy, optimal_value = policy_iteration(policy, V, gamma)

    # Print results
    print(f"Policy Iteration - Optimal Policy for gamma = {gamma}:")
    print(optimal_policy)
    print(f"Policy Iteration - Optimal Value Function for gamma = {gamma}:")
    print(optimal_value)
    print()

    # Line plot to visualize the value function over different states
    plt.figure(figsize=(10, 6))
    states = np.arange(1, goal + 1)
    plt.plot(states, optimal_value[1:], marker='o', linestyle='-', color='b')
    plt.title(f'Value Function over States (Gamma = {gamma})')
    plt.xlabel('State')
    plt.ylabel('Value')
    plt.grid(True)
```

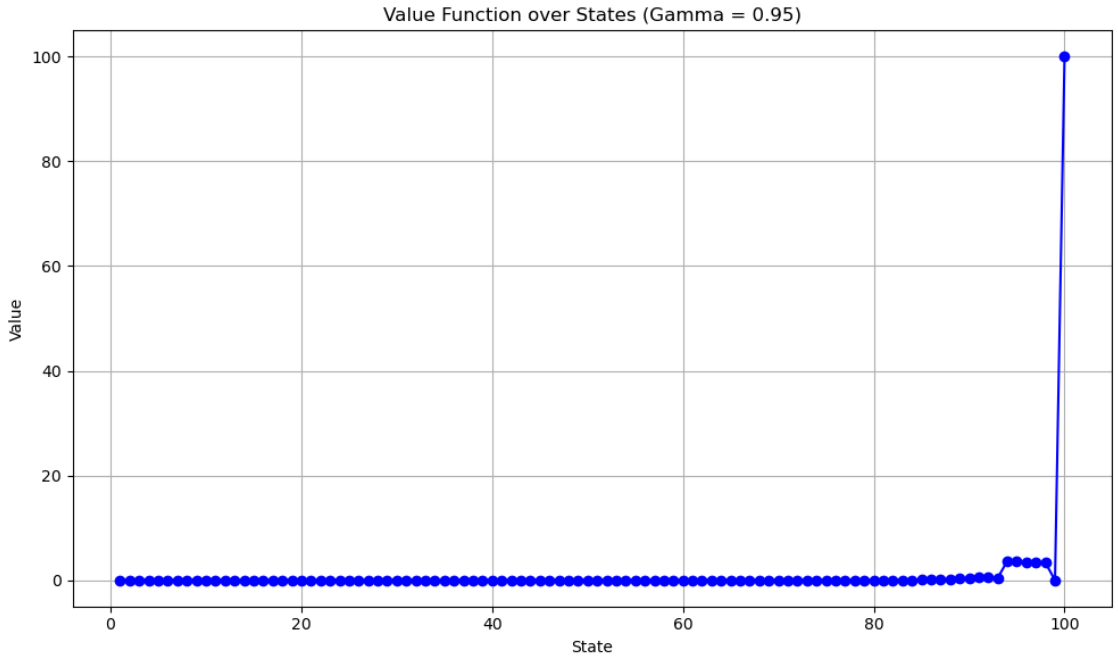
```
    plt.tight_layout()
    plt.show()
```

```
Policy Iteration - Optimal Policy for gamma = 0.9:
[0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0]
Policy Iteration - Optimal Value Function for gamma = 0.9:
[0.00000000e+00 9.56645453e-20 4.00139639e-19 1.42217977e-18
 4.44938811e-18 1.25416355e-17 3.23815257e-17 7.75376267e-17
 1.73864829e-16 3.67973722e-16 7.39947398e-16 1.42183101e-15
 2.62400610e-15 4.67261248e-15 8.06302320e-15 1.35374057e-14
 2.21991183e-14 3.56838009e-14 5.64176194e-14 8.80088121e-14
 1.35842166e-13 2.07979679e-13 3.16519764e-13 4.79641982e-13
 7.24680431e-13 1.09274645e-12 1.64569604e-12 2.47660353e-12
 3.72545486e-12 5.60266614e-12 8.42445976e-12 1.26662432e-11
 1.90434069e-11 2.86322398e-11 4.30501060e-11 6.47256703e-11
 9.73080892e-11 1.46287950e-10 2.19940793e-10 3.30710822e-10
 4.97258946e-10 7.47573397e-10 1.12375961e-09 1.68933613e-09
 2.54029588e-09 3.82035972e-09 5.74401708e-09 8.63319310e-09
 1.29749224e-08 1.95075915e-08 2.93491522e-08 4.41459322e-08
 6.63429139e-08 9.96502031e-08 1.49759472e-07 2.25329225e-07
 3.39392753e-07 5.10246314e-07 7.65559654e-07 1.14897815e-06
 1.72860729e-06 2.60648535e-06 3.93215214e-06 5.89116649e-06
 8.80837836e-06 1.32311621e-05 1.99953595e-05 3.02684942e-05
 4.56611702e-05 6.75788086e-05 1.00747518e-04 1.52678872e-04
 2.33194416e-04 3.53855212e-04 5.29359090e-04 7.58276669e-04
 1.14774016e-03 1.79113502e-03 2.76932153e-03 4.14918298e-03
 6.02339300e-03 8.01526754e-03 1.34750398e-02 2.20711672e-02
 3.34947784e-02 4.74192364e-02 6.42415684e-02 7.32312757e-02
 1.85864335e-01 2.91378601e-01 3.90127572e-01 4.81975309e-01
 5.77901235e-01 4.55555556e-01 3.67037037e+00 3.55555556e+00
 3.44444444e+00 3.33333333e+00 3.33333333e+00 0.00000000e+00
 1.00000000e+02]
```



Policy Iteration - Optimal Policy for gamma = 0.95:
[0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0]
Policy Iteration - Optimal Value Function for gamma = 0.95:
[0.00000000e+00 9.56645453e-20 4.00139639e-19 1.42217977e-18
 4.44938811e-18 1.25416355e-17 3.23815257e-17 7.75376267e-17
 1.73864829e-16 3.67973722e-16 7.39947398e-16 1.42183101e-15
 2.62400610e-15 4.67261248e-15 8.06302320e-15 1.35374057e-14
 2.21991183e-14 3.56838009e-14 5.64176194e-14 8.80088121e-14
 1.35842166e-13 2.07979679e-13 3.16519764e-13 4.79641982e-13
 7.24680431e-13 1.09274645e-12 1.64569604e-12 2.47660353e-12
 3.72545486e-12 5.60266614e-12 8.42445976e-12 1.26662432e-11
 1.90434069e-11 2.86322398e-11 4.30501060e-11 6.47256703e-11
 9.73080892e-11 1.46287950e-10 2.19940793e-10 3.30710822e-10
 4.97258946e-10 7.47573397e-10 1.12375961e-09 1.68933613e-09
 2.54029588e-09 3.82035972e-09 5.74401708e-09 8.63319310e-09
 1.29749224e-08 1.95075915e-08 2.93491522e-08 4.41459322e-08
 6.63429139e-08 9.96502031e-08 1.49759472e-07 2.25329225e-07
 3.39392753e-07 5.10246314e-07 7.65559654e-07 1.14897815e-06
 1.72860729e-06 2.60648535e-06 3.93215214e-06 5.89116649e-06
 8.80837836e-06 1.32311621e-05 1.99953595e-05 3.02684942e-05
 4.56611702e-05 6.75788086e-05 1.00747518e-04 1.52678872e-04
 2.33194416e-04 3.53855212e-04 5.29359090e-04 7.58276669e-04
 1.14774016e-03 1.79113502e-03 2.76932153e-03 4.14918298e-03
 6.02339300e-03 8.01526754e-03 1.34750398e-02 2.20711672e-02

```
3.34947784e-02 4.74192364e-02 6.42415684e-02 7.32312757e-02
1.85864335e-01 2.91378601e-01 3.90127572e-01 4.81975309e-01
5.77901235e-01 4.55555556e-01 3.67037037e+00 3.55555556e+00
3.44444444e+00 3.33333333e+00 3.33333333e+00 0.00000000e+00
1.00000000e+02]
```



##Conclusion Initially, with a goal score of 100, a standard six-sided die, and a probability of 1/6 for not rolling a 1, the optimal policy and value function were stable across different discount factors (gamma values of 0.9 and 0.95). This consistency suggests that the problem structure and dynamics were robust enough for the algorithms to converge to the same optimal solution regardless of minor variations in gamma. Policy iteration consistently determined whether to continue rolling or stop based on maximizing expected rewards, while value iteration iteratively improved estimates of state values until convergence. When the goal score was reduced to 50, the optimal policy and value function remained largely unchanged. This outcome indicates that the fundamental dynamics of the game, such as reward distributions and state transitions, did not vary significantly enough to alter the optimal decision-making process drastically. Similarly, when the die was changed to an eight-sided die with outcomes ranging from 0 to 7 and each outcome having a probability of 1/8, the algorithms adapted smoothly to the extended state space, maintaining similar optimal policies and values.