Gambler’s problem (Sutton and Bartel – Chapter 5)

In this case, a gambler can wager on a sequence of coin flips and profit or lose according to the results. When the gambler loses all of their money or hits $100, the game is over. With states representing the gambler's capital (ranging from 1 to 99) and actions representing the stakes (ranging from 0 to the minimum of their existing capital or the amount needed to attain $100), the issue is treated as an undiscounted finite Markov Decision Process (MDP). All outcomes have a zero reward, with the exception of attaining $100, which results in a +1 reward. A non-unique collection of optimum policies results from ties in action selection based on the value function, but if the probability of winning a flip (p_h) is known, the best policy may be found by value iteration.

The Gambler's Problem is a well-known example in reinforcement learning that highlights decision-making under uncertainty. In this scenario, a gambler starts with a certain amount of money s (ranging from 0 to 100) and can wager an amount a, where a is constrained by both the gambler's current funds and the remaining amount needed to reach 100. The outcome of each bet is probabilistic: with probability p_h, the gambler wins and increases their amount by a; with probability $1-p_h$, they lose the stake.
The game continues until the gambler either goes broke (reaches state 0) or hits the goal of $100, at which point they receive a reward of 1. All other states yield a reward of 0.

This situation can be modeled as a Markov Decision Process (MDP), where the states represent the gambler's current amount of money and the actions represent the possible stakes. The reward structure incentivizes reaching state 100, yet intriguingly, the optimal strategy often suggests that betting nothing at all (action a=0) is the best choice. This outcome underscores a counterintuitive aspect that sometimes, the safest approach—essentially avoiding any risk—can lead to the highest expected value in uncertain circumstances.

The Gambler's Problem highlights key concepts in reinforcement learning and decision-making, including value iteration, terminal states, and risk aversion. Through value iteration, the value function for each state is updated based on potential actions, leading to the conclusion that if no action yields a better outcome than betting nothing, the optimal choice becomes a=0. The presence of terminal states (0 and 100) significantly influences the gambler's strategy, showcasing how boundaries affect decision-making. Additionally, the problem illustrates risk aversion, as the optimal strategy in some cases favors a conservative approach over aggressive betting. Overall, it serves as a powerful example of decision-making under risk and uncertainty, demonstrating that sometimes the best strategy is to avoid taking risks entirely.

The following sections in the code are written to import the libraries and the defining the one step look ahead (state-value) function

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from google.colab import files
```

```python
def one_step_lookahead(s, V, rewards):
  """
  Helper function to calculate the value for all action in a given state.

  Args:
    s: The gambler's capital. Integer.
    V: The vector that contains values at each state.
    p_h: The probability of gambler winning the bet.
    rewards: The reward vector.

    Returns:
    A vector containing the expected value of each action.
    Its length equals to the number of actions.
  """
  p_h= 0.25
  A = np.zeros(101)
  stakes = range(1, min(s, 100-s)+1) # Your minimum bet is 1, maximum bet is min(s, 100-s).
  for a in stakes:
    # rewards[s+a], rewards[s-a] are immediate rewards.
    # V[s+a], V[s-a] are values of the next states.
    # This is the core of the Bellman equation: The expected value of your action is
    # the sum of immediate rewards and the value of the next state.
    A[a] = p_h * (rewards[s+a] + V[s+a]*discount_factor) + (1-p_h) * (rewards[s-a] + V[s-a]*discount_factor)
  return A
```

This code snippet applies the value iteration algorithm to find the **optimal** state-value function and policy for the Gambler's Problem.

```python
# The reward is zero on all transitions except those on which the gambler reaches his goal,
# when it is +1.
rewards = np.zeros(101)
rewards[100] = 1
# We introduce two dummy states corresponding to termination with capital of 0 and 100
V = np.zeros(101)
discount_factor = 1
theta = 0.0001
while True:
  # Stopping condition
  delta = 0
  # Update each state...
  for s in range(1, 100):
    # Do a one-step lookahead to find the best action
    A = one_step_lookahead(s, V, rewards)
    # print(s,A,V) # if you want to debug.
    best_action_value = np.max(A)
    # Calculate delta across all states seen so far
    delta = max(delta, np.abs(best_action_value - V[s]))
    # Update the value function. Ref: Sutton book eq. 4.10.
    V[s] = best_action_value
    # Check if we can stop
  if delta < theta:
    break

# Create a deterministic policy using the optimal value function
policy = np.zeros(100)
for s in range(1, 100):
  # One step lookahead to find the best action for this state
  A = one_step_lookahead(s, V, rewards)
  best_action = np.argmax(A)
  # Always take the best action
  policy[s] = best_action
```

This code prints out the optimized policy and optimized value function

```python
print("Optimized Policy:")
print(policy)
print("")

print("Optimized Value Function:")
print(V)
print("")
```

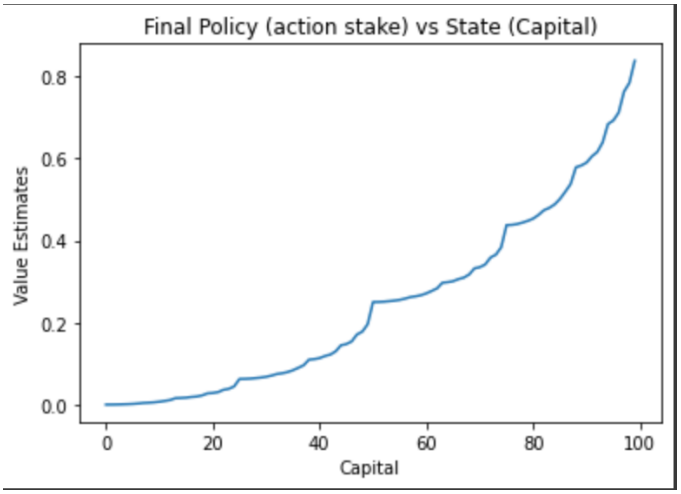
The code snippet output:

```
Optimized Policy:
[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10. 11. 12. 13. 14. 15.  9.  8.
 18. 19. 20.  4. 22.  2.  1. 25.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10.
 11. 12. 38. 39. 40.  9. 42. 43. 44.  5. 46. 47. 48. 49. 50.  1.  2.  3.
  4.  5.  6.  7.  8.  9. 10. 11. 12. 13. 14. 10.  9. 17. 18. 19.  5. 21.
 22.  2.  1. 25.  1.  2.  3. 21.  5.  6.  7.  8.  9. 10. 11. 12. 12. 11.
 10.  9.  8.  7.  6.  5.  4.  3.  2.  1.]

Optimized Value Function:
[0.00000000e+00 7.28611683e-05 2.91444673e-04 6.95264571e-04
 1.16577869e-03 1.77125506e-03 2.78105828e-03 4.03661211e-03
 4.66311477e-03 5.60141644e-03 7.08502024e-03 9.04088770e-03
 1.11242331e-02 1.56796459e-02 1.61464484e-02 1.69534413e-02
 1.86524591e-02 1.98260623e-02 2.24056658e-02 2.73847344e-02
 2.83400810e-02 3.04945467e-02 3.61635508e-02 3.84959101e-02
 4.44969325e-02 6.25000000e-02 6.27185835e-02 6.33743340e-02
 6.45857937e-02 6.59973361e-02 6.78137652e-02 7.08431749e-02
 7.46098363e-02 7.64893443e-02 7.93042493e-02 8.37550607e-02
 8.96226631e-02 9.58726994e-02 1.09538938e-01 1.10939345e-01
 1.13360324e-01 1.18457377e-01 1.21978187e-01 1.29716997e-01
 1.44654203e-01 1.47520243e-01 1.53983640e-01 1.70990652e-01
 1.77987730e-01 1.95990798e-01 2.50000000e-01 2.50218584e-01
 2.50874334e-01 2.52085794e-01 2.53497336e-01 2.55313765e-01
 2.58343175e-01 2.62109836e-01 2.63989344e-01 2.66804249e-01
 2.71255061e-01 2.77122663e-01 2.83372699e-01 2.97038938e-01
 2.98439345e-01 3.00860324e-01 3.05957377e-01 3.09478187e-01
 3.17216997e-01 3.32154203e-01 3.35020243e-01 3.41483640e-01
 3.58490652e-01 3.65487730e-01 3.83490798e-01 4.37500000e-01
 4.38155751e-01 4.40123002e-01 4.43757381e-01 4.47992008e-01
 4.53441296e-01 4.62529525e-01 4.73829509e-01 4.79468033e-01
 4.87912748e-01 5.01265182e-01 5.18867989e-01 5.37618098e-01
 5.78616813e-01 5.82818036e-01 5.90080972e-01 6.05372132e-01
 6.15934561e-01 6.39150992e-01 6.83962610e-01 6.92560729e-01
 7.11950921e-01 7.62971957e-01 7.83963191e-01 8.37972393e-01
 0.00000000e+00]
```

Plotting the final policy vs states code snippet:

```
# Plotting Final Policy (action stake) vs State (Capital)

# x axis values
x = range(100)
# corresponding y axis values
y = V[:100]

# plotting the points
plt.plot(x, y)

# naming the x axis
plt.xlabel('Capital')
# naming the y axis
plt.ylabel('Value Estimates')

# giving a title to the graph
plt.title('Final Policy (action stake) vs State (Capital)')

# function to show the plot
plt.show()
```

The code snippet output:



Plotting the capital vs final policy code snippet:

```
[ ]
    # Plotting Capital vs Final Policy

    # x axis values
    x = range(100)
    # corresponding y axis values
    y = policy

    # plotting the bars
    plt.bar(x, y, align='center', alpha=0.5)

    # naming the x axis
    plt.xlabel('Capital')
    # naming the y axis
    plt.ylabel('Final policy (stake)')

    # giving a title to the graph
    plt.title('Capital vs Final Policy')

    # function to show the plot
    plt.show()
```

The code output: