```
# value iteration example
#We have a 4×4 Gridworld where an agent moves in one of four directions: up, down, left, or right.
#The goal is to reach the terminal state (bottom-right corner) while maximizing cumulative rewards.
#The agent starts at any state (except the goal).
#Every move incurs a small penalty (-0.04), to encourage shorter paths.
#The agent cannot move outside the grid (it stays in place if it tries).
#The goal state (bottom-right corner) gives a reward of 1 and is terminal.
#The agent wants to find the best path to the goal.

```

```
import numpy as np

# Grid World parameters
grid_size = 4 # you can increase the grid size to increase the complexity of the problem
num_states = grid_size * grid_size
gamma = 0.9  # Discount factor
theta = 1e-4  # Convergence threshold
step_cost = -0.04  # Small penalty per move

# Actions and transitions
actions = ["UP", "DOWN", "LEFT", "RIGHT"]
action_deltas = {"UP": -grid_size, "DOWN": grid_size, "LEFT": -1, "RIGHT": 1}

# Initialize Value Function
V = np.zeros(num_states)
V[-1] = 1.0  # ✅ Ensure goal state starts with a value of 1.0

# Define rewards
rewards = np.full(num_states, step_cost)  # Small cost everywhere
rewards[-1] = 1  # Goal state reward

print("Initial Rewards:")
print(rewards.reshape((grid_size, grid_size)))

# Value Iteration Algorithm
iteration = 0
while True:
    delta = 0
    V_new = np.copy(V)

    print(f"\nIteration {iteration}:")
    print(np.round(V.reshape((grid_size, grid_size)), 2))

    for s in range(num_states):
        if s == num_states - 1:  # Goal state remains unchanged
            continue

        max_value = float('-inf')

        for action in actions:
            s_next = s + action_deltas[action]

            # Ensure valid moves
            if s_next < 0 or s_next >= num_states:
                continue
            if action == "LEFT" and s % grid_size == 0:
                continue
            if action == "RIGHT" and (s + 1) % grid_size == 0:
                continue

            # ✅ Correct Bellman Update: Max over possible actions
            value = rewards[s] + gamma * V[s_next]
            max_value = max(max_value, value)

        V_new[s] = max_value
        delta = max(delta, abs(V_new[s] - V[s]))

    V = V_new
    iteration += 1
    if delta < theta:
        break  # Convergence

# Final Value Function
print("\nFinal Optimal Value Function:")
print(np.round(V.reshape((grid_size, grid_size)), 2))
```

```
Initial Rewards:
[[-0.04 -0.04 -0.04 -0.04]
 [-0.04 -0.04 -0.04 -0.04]
 [-0.04 -0.04 -0.04 -0.04]
 [-0.04 -0.04 -0.04  1.  ]]

Iteration 0:
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 1.]]

Iteration 1:
[[-0.04 -0.04 -0.04 -0.04]
 [-0.04 -0.04 -0.04 -0.04]
 [-0.04 -0.04 -0.04  0.86]
 [-0.04 -0.04  0.86  1.  ]]

Iteration 2:
[[-0.08 -0.08 -0.08 -0.08]
 [-0.08 -0.08 -0.08  0.73]
 [-0.08 -0.08  0.73  0.86]
 [-0.08  0.73  0.86  1.  ]]

Iteration 3:
[[-0.11 -0.11 -0.11  0.62]
 [-0.11 -0.11  0.62  0.73]
 [-0.11  0.62  0.73  0.86]
 [ 0.62  0.73  0.86  1.  ]]

Iteration 4:
[[-0.14 -0.14  0.52  0.62]
 [-0.14  0.52  0.62  0.73]
 [ 0.52  0.62  0.73  0.86]
 [ 0.62  0.73  0.86  1.  ]]

Iteration 5:
[[-0.16  0.43  0.52  0.62]
 [ 0.43  0.52  0.62  0.73]
 [ 0.52  0.62  0.73  0.86]
 [ 0.62  0.73  0.86  1.  ]]

Iteration 6:
[[0.34 0.43 0.52 0.62]
 [0.43 0.52 0.62 0.73]
 [0.52 0.62 0.73 0.86]
 [0.62 0.73 0.86 1.  ]]

Final Optimal Value Function:
[[0.34 0.43 0.52 0.62]
 [0.43 0.52 0.62 0.73]
 [0.52 0.62 0.73 0.86]
 [0.62 0.73 0.86 1.  ]]
```

```python
# policy iteration for the above example
import numpy as np

# Grid World parameters
grid_size = 4  # Can be increased to increase complexity
num_states = grid_size * grid_size
gamma = 0.9  # Discount factor
theta = 1e-4  # Convergence threshold
step_cost = -0.04  # Small penalty per move

# Actions and transitions
actions = ["UP", "DOWN", "LEFT", "RIGHT"]
action_deltas = {"UP": -grid_size, "DOWN": grid_size, "LEFT": -1, "RIGHT": 1}

# Initialize Value Function and Policy
V = np.zeros(num_states)
policy = np.random.choice(actions, size=num_states)  # Random initial policy
V[-1] = 1.0  # Goal state has value 1.0

# Define rewards
rewards = np.full(num_states, step_cost)
rewards[-1] = 1  # Goal state reward

print("Initial Policy:")
print(policy.reshape((grid_size, grid_size)))
```

```python
# Policy Iteration
policy_stable = False
iteration = 0

while not policy_stable:
    # Policy Evaluation
    while True:
        delta = 0
        V_new = np.copy(V)

        for s in range(num_states):
            if s == num_states - 1:  # Skip goal state
                continue

            action = policy[s]
            s_next = s + action_deltas[action]

            # Ensure valid moves
            if s_next < 0 or s_next >= num_states:
                continue
            if action == "LEFT" and s % grid_size == 0:
                continue
            if action == "RIGHT" and (s + 1) % grid_size == 0:
                continue

            # Bellman Equation for Policy Evaluation
            V_new[s] = rewards[s] + gamma * V[s_next]
            delta = max(delta, abs(V_new[s] - V[s]))

        V = V_new
        if delta < theta:
            break  # Convergence

    # Policy Improvement
    policy_stable = True
    for s in range(num_states):
        if s == num_states - 1:  # Skip goal state
            continue

        best_action = None
        best_value = float('-inf')

        for action in actions:
            s_next = s + action_deltas[action]

            # Ensure valid moves
            if s_next < 0 or s_next >= num_states:
                continue
            if action == "LEFT" and s % grid_size == 0:
                continue
            if action == "RIGHT" and (s + 1) % grid_size == 0:
                continue

            value = rewards[s] + gamma * V[s_next]
            if value > best_value:
                best_value = value
                best_action = action

        if policy[s] != best_action:
            policy_stable = False  # Policy changed, need another iteration
            policy[s] = best_action

    iteration += 1
    print(f"\nIteration {iteration}:")
    print("Value Function:")
    print(np.round(V.reshape((grid_size, grid_size)), 2))
    print("Policy:")
    print(policy.reshape((grid_size, grid_size)))

print("\nFinal Optimal Value Function:")
print(np.round(V.reshape((grid_size, grid_size)), 2))

print("\nFinal Optimal Policy:")
print(policy.reshape((grid_size, grid_size)))
```

```
Initial Policy:
[['UP' 'DOWN' 'RIGHT' 'LEFT']
 ['UP' 'UP' 'DOWN' 'RIGHT']
 ['DOWN' 'LEFT' 'RIGHT' 'RIGHT']
 ['UP' 'LEFT' 'RIGHT' 'RIGHT']]

Iteration 1:
Value Function:
[[ 0.   -0.4  -0.4  -0.4 ]
 [-0.04 -0.4  -0.08  0.  ]
 [-0.4  -0.4  -0.04  0.  ]
 [-0.4  -0.4   0.86  1.  ]]
Policy:
[['DOWN' 'LEFT' 'DOWN' 'DOWN']
 ['UP' 'LEFT' 'RIGHT' 'DOWN']
 ['UP' 'RIGHT' 'DOWN' 'DOWN']
 ['UP' 'RIGHT' 'RIGHT' 'RIGHT']]

Iteration 2:
Value Function:
[[-0.4  -0.4   0.52  0.62]
 [-0.4  -0.4   0.62  0.73]
 [-0.4   0.62  0.73  0.86]
 [-0.4   0.73  0.86  1.  ]]
Policy:
[['DOWN' 'RIGHT' 'DOWN' 'DOWN']
 ['UP' 'DOWN' 'DOWN' 'DOWN']
 ['RIGHT' 'DOWN' 'DOWN' 'DOWN']
 ['RIGHT' 'RIGHT' 'RIGHT' 'RIGHT']]

Iteration 3:
Value Function:
[[-0.4   0.43  0.52  0.62]
 [-0.4   0.52  0.62  0.73]
 [ 0.52  0.62  0.73  0.86]
 [ 0.62  0.73  0.86  1.  ]]
Policy:
[['RIGHT' 'DOWN' 'DOWN' 'DOWN']
 ['DOWN' 'DOWN' 'DOWN' 'DOWN']
 ['DOWN' 'DOWN' 'DOWN' 'DOWN']
 ['RIGHT' 'RIGHT' 'RIGHT' 'RIGHT']]

Iteration 4:
Value Function:
[[0.34 0.43 0.52 0.62]
 [0.43 0.52 0.62 0.73]
 [0.52 0.62 0.73 0.86]
 [0.62 0.73 0.86 1.  ]]
Policy:
[['DOWN' 'DOWN' 'DOWN' 'DOWN']
 ['DOWN' 'DOWN' 'DOWN' 'DOWN']
 ['DOWN' 'DOWN' 'DOWN' 'DOWN']
 ['RIGHT' 'RIGHT' 'RIGHT' 'RIGHT']]

Iteration 5:
Value Function:
[[0.34 0.43 0.52 0.62]
 [0.43 0.52 0.62 0.73]
```