In [1]:
```
import sympy as sp
import numpy as np


x, u, a, b, c = sp.symbols('x u a b c')


gamma = sp.Rational(1, 2)
sqrt2 = sp.sqrt(2)

# stage cost
stage_cost = sp.Rational(1, 2) * x**2 + sp.Rational(1, 2) * u**2

# expected value of V*
E_x_next_squared = 2*x**2 + 2*sqrt2*x*u + u**2 + 2
E_x_next = sqrt2*x + u
E_V_next = a*E_x_next_squared + b*E_x_next + c

# Bellman equation RHS (before minimization)
bellman_rhs = stage_cost + gamma * E_V_next

# find optimal u by taking derivative and setting to zero
du_bellman = sp.diff(bellman_rhs, u)
u_star = sp.solve(du_bellman, u)[0]


# substitute back into bellman equation
bellman_rhs_optimal = bellman_rhs.subs(u, u_star)
bellman_rhs_optimal_simplified = sp.simplify(bellman_rhs_optimal)
bellman_expanded = sp.expand(bellman_rhs_optimal_simplified)
bellman_collected = sp.collect(bellman_expanded, x)
coeff_x2 = bellman_collected.coeff(x, 2)
coeff_x1 = bellman_collected.coeff(x, 1)
coeff_x0 = bellman_collected.coeff(x, 0)



# set up equations by matching coefficients
eq1 = sp.Eq(a, coeff_x2)
eq2 = sp.Eq(b, coeff_x1)
eq3 = sp.Eq(c, coeff_x0)



# solve the system of equations
# first solve for a from eq1
a_solutions = sp.solve(eq1, a)

# choose positive solution for a
a_val = 1  # From (2a + 1)(a - 1) = 0, we choose a = 1


# substitute a = 1 into eq2 to find b
eq2_with_a = eq2.subs(a, a_val)
b_val = sp.solve(eq2_with_a, b)[0]


# substitute a = 1 and b = 0 into eq3 to find c
```

```
eq3_with_a_b = eq3.subs([(a, a_val), (b, b_val)])
c_val = sp.solve(eq3_with_a_b, c)[0]


print("FINAL SOLUTION:")
print(f"a = {a_val}")
print(f"b = {b_val}")
print(f"c = {c_val}")
print(f"V*(x) = {a_val}x^2 + {b_val}x + {c_val}")
```

```
FINAL SOLUTION:
a = 1
b = 0
c = 2
V*(x) = 1x^2 + 0x + 2
```

In [6]:
```
import numpy as np

# Transition probability matrices
PA = np.array([[0.1, 0.7, 0.2],
               [0.5, 0.3, 0.2],
               [0.0, 0.0, 1.0]])

PB = np.array([[0.3, 0.5, 0.2],
               [0.5, 0.3, 0.2],
               [0.0, 0.0, 1.0]])

# State and control spaces
STATE = np.array([1, 2, 3])
CONTROL = ['a', 'b']
TERMINAL_STATE = 3
TERMINAL_COST = 0

def get_stage_cost(state, control):
    """Get the stage cost for a given state and control"""
    if state == TERMINAL_STATE:
        return 0
    else:
        if control == 'a':
            return 16 * state
        else:
            return 5 * state

def get_transition_prob(state, control, next_state):
    """Get transition probability P(next_state | state, control)"""
    state_index = state - 1  # Convert to 0-based index
    next_state_index = next_state - 1
    if control == 'a':
        return PA[state_index, next_state_index]
    else:
        return PB[state_index, next_state_index]

def get_updated_value(state, value_func, gamma=0.8):
    """Compute the updated value for a single state using Bellman operator"""
    if state == TERMINAL_STATE:
        return TERMINAL_COST

    min_value = np.inf
    for control in CONTROL:
        # Compute expected value for this control
```

```
        expected_value = get_stage_cost(state, control)
        for next_state in STATE:
            prob = get_transition_prob(state, control, next_state)
            expected_value += gamma * prob * value_func[next_state - 1]

        # Keep track of minimum value across all controls
        if expected_value < min_value:
            min_value = expected_value

    return min_value

def value_iteration(initial_value, gamma=0.8, max_iter=1):
    """Perform value iteration for specified number of iterations"""
    value = initial_value.copy()

    for iteration in range(max_iter):
        new_value = value.copy()

        # Update values for all non-terminal states
        for state in STATE:
            if state != TERMINAL_STATE:
                new_value[state - 1] = get_updated_value(state, value, gamma)

        # Update value function
        value = new_value

        print(f"After iteration {iteration + 1}:")
        print(f"V(1) = {value[0]:.2f}")
        print(f"V(2) = {value[1]:.2f}")
        print(f"V(3) = {value[2]:.2f}")
        print()

    return value

# Initial value function
V0 = np.array([20.0, 10.0, 0.0])

print("Initial value function:")
print(f"V0(1) = {V0[0]}")
print(f"V0(2) = {V0[1]}")
print(f"V0(3) = {V0[2]}")
print()

# Perform one iteration of value iteration
V1 = value_iteration(V0, gamma=0.8, max_iter=1)


print("Final result after one iteration:")
print(f"V1 = [{V1[0]:.1f}, {V1[1]:.1f}, {V1[2]:.1f}]")
```

```
Initial value function:
V0(1) = 20.0
V0(2) = 10.0
V0(3) = 0.0

After iteration 1:
V(1) = 13.80
V(2) = 20.40
V(3) = 0.00

Final result after one iteration:
V1 = [13.8, 20.4, 0.0]
```

In [ ]:

```python
import numpy as np
import cvxpy as cp

# params
n_control = 2
n_state = 2

# weights positive
w = np.random.rand(2)

# Problem parameters
gamma = 0.8

# Transition probability matrices
P_a = np.array([[1/8, 7/8],
                [5/8, 3/8]])

P_b = np.array([[3/8, 5/8],
                [5/8, 3/8]])
identity = np.eye(n_state)

# stage cost
cost_a = np.array([16., 32.])
cost_b = np.array([5., 10.])

# decistion variables
value = cp.Variable(n_state)

# LP objective
objective = cp.Maximize(w.T @ value)

# LP constraints
constraints = []
# control a
constraints.append((identity - gamma * P_a) @ value <= cost_a)
# control b
constraints.append((identity - gamma * P_b) @ value <= cost_b)

# solve
problem = cp.Problem(objective, constraints)
problem.solve()
print(f"Optimal value: {value.value}")
```

```
Optimal value: [35.41666674 39.58333341]
```

In [ ]: