DRL_GROUP_244_Dynamic_Programming_code_Assignment01

June 21, 2025

0.0.1 Group ID: 244

1 Group members

Name

Email

Student ID

Contribution

G. Ankur Vatsa

2023aa05727@wilp.bits-pilani.ac.in

2023aa05727

100%

DURGA PRASAD YADAV

2024ab05147@wilp.bits-pilani.ac.in

2024ab05147

100%

JAIDEEP PALIT

2024aa05319@wilp.bits-pilani.ac.in

2024aa05319

100%

FIYAS AHAMED A

2023aa05796@wilp.bits-pilani.ac.in

2023aa05796

100%

2 Considerations

The classes are created first, the main function is run at the end as final solution.

3 1. Custom Environment Creation (SmartSupplierEnv)

3.1 1.1 Required Libraries and Dependencies

- **numpy**: For numerical computations, array operations, and mathematical functions
- **random**: For generating random numbers (market state transitions and simulation)
- **typing**: For type hints to improve code readability and maintainability
- **dataclasses**: For creating clean, immutable data structures (State representation)
- **enum**: For defining enumerated types (MarketState and Action)
- **matplotlib & seaborn**: For data visualization and plotting results

[47]:
```
!pip install numpy matplotlib seaborn
```

```
Requirement already satisfied: numpy in ./.venv/lib/python3.9/site-packages
(2.0.2)
Requirement already satisfied: matplotlib in ./.venv/lib/python3.9/site-packages
(3.9.4)
Requirement already satisfied: seaborn in ./.venv/lib/python3.9/site-packages
(0.13.2)
Requirement already satisfied: contourpy>=1.0.1 in ./.venv/lib/python3.9/site-
packages (from matplotlib) (1.3.0)
Requirement already satisfied: cycler>=0.10 in ./.venv/lib/python3.9/site-
packages (from matplotlib) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in ./.venv/lib/python3.9/site-
packages (from matplotlib) (4.58.4)
Requirement already satisfied: kiwisolver>=1.3.1 in ./.venv/lib/python3.9/site-
packages (from matplotlib) (1.4.7)
Requirement already satisfied: packaging>=20.0 in ./.venv/lib/python3.9/site-
packages (from matplotlib) (25.0)
Requirement already satisfied: pillow>=8 in ./.venv/lib/python3.9/site-packages
(from matplotlib) (11.2.1)
Requirement already satisfied: pyparsing>=2.3.1 in ./.venv/lib/python3.9/site-
packages (from matplotlib) (3.2.3)
Requirement already satisfied: python-dateutil>=2.7 in
./.venv/lib/python3.9/site-packages (from matplotlib) (2.9.0.post0)
Requirement already satisfied: importlib-resources>=3.2.0 in
./.venv/lib/python3.9/site-packages (from matplotlib) (6.5.2)
Requirement already satisfied: pandas>=1.2 in ./.venv/lib/python3.9/site-
packages (from seaborn) (2.3.0)
Requirement already satisfied: zipp>=3.1.0 in ./.venv/lib/python3.9/site-
packages (from importlib-resources>=3.2.0->matplotlib) (3.23.0)
Requirement already satisfied: pytz>=2020.1 in ./.venv/lib/python3.9/site-
packages (from pandas>=1.2->seaborn) (2025.2)
Requirement already satisfied: tzdata>=2022.7 in ./.venv/lib/python3.9/site-
```

```
packages (from pandas>=1.2->seaborn) (2025.2)
Requirement already satisfied: six>=1.5 in ./.venv/lib/python3.9/site-packages
(from python-dateutil>=2.7->matplotlib) (1.17.0)
```

[48]:
```python
import numpy as np
import random
from typing import Dict, List, Tuple, Optional
from dataclasses import dataclass
from enum import Enum
import matplotlib.pyplot as plt
import seaborn as sns

# Set random seed for reproducibility
np.random.seed(42)
random.seed(42)
```

3.2 1.2 Market State Definition

Enum **MarketState** represents two possible market conditions that affect product pricing in Smart Supplier problem. This crucial component of our state space captures market uncertainty.

3.2.1 Market Dynamics:

The market alternates between two states with equal probability (50% each day):

1. **HIGH_DEMAND_A (Market State 1)**:
   - **Product A**: $8 per unit (highly profitable)
   - **Product B**: $2 per unit (low profit)
   - **Strategy Implication**: Favor producing Product A when possible
2. **HIGH_DEMAND_B (Market State 2)**:
   - **Product A**: $3 per unit (low profit)

   - **Product B**: $5 per unit (moderately profitable)
   - **Strategy Implication**: Favor producing Product B when possible

3.2.2 Key Design Decisions:

- **Stochastic Transitions**: Market state changes randomly each day (Markovian property)
- **Equal Probabilities**: Each state has 50% chance to maintain realism
- **Significant Price Differences**: Creates clear incentives for adaptive strategies
- **Type Safety**: Using Enum prevents invalid market state assignments

[49]:
```python
# Define market states and their product prices
class MarketState(Enum):
    """
    MARKET STATE ENUMERATION
    =======================


    Represents the two possible market conditions that affect product prices:
```

```
    HIGH_DEMAND_A (State 1): Market favors Product A
    - Product A: $8 per unit (high profit)
    - Product B: $2 per unit (low profit)

    HIGH_DEMAND_B (State 2): Market favors Product B
    - Product A: $3 per unit (low profit)
    - Product B: $5 per unit (high profit)

    This enum provides type safety and clear naming for market conditions.
    """
    HIGH_DEMAND_A = 1
    HIGH_DEMAND_B = 2
```

3.3 1.3 Action Space Definition

The **Action** enum defines all possible production decisions the Smart Supplier can make each day. This discrete action space is carefully designed to respect resource constraints while providing meaningful strategic choices.

3.3.1 Resource Constraints:

- **Daily Raw Materials**: 10 units available each day
- **Product A Cost**: 2 raw materials per unit
- **Product B Cost**: 1 raw material per unit
- **Constraint**: Total consumption 10 raw materials per day

3.3.2 Available Actions:

1. **PRODUCE_2A_0B**: Produce 2 units of A, 0 units of B
   - **Cost**: $2\times2 + 0\times1 = 4$ raw materials
   - **Strategy**: Focus on high-value Product A when market favors it
2. **PRODUCE_1A_2B**: Produce 1 unit of A, 2 units of B
   - **Cost**: $1\times2 + 2\times1 = 4$ raw materials
   - **Strategy**: Balanced production for mixed market conditions
3. **PRODUCE_0A_5B**: Produce 0 units of A, 5 units of B
   - **Cost**: $0\times2 + 5\times1 = 5$ raw materials

   - **Strategy**: Focus on Product B when market favors it
4. **PRODUCE_3A_0B**: Produce 3 units of A, 0 units of B
   - **Cost**: $3\times2 + 0\times1 = 6$ raw materials
   - **Strategy**: Aggressive Product A production in favorable markets
5. **DO_NOTHING**: Produce 0 units of both products
   - **Cost**: $0\times2 + 0\times1 = 0$ raw materials
   - **Strategy**: Conservative approach or when resources are insufficient

3.3.3 Design Rationale:

- **Discrete Actions**: Simplifies learning and ensures tractable state space

- **Resource Validation**: All actions respect the 10 RM daily limit
- **Strategic Diversity**: Actions range from conservative to aggressive
- **Market Adaptation**: Different actions optimal under different market conditions

[50]:
```python
# Define actions: (num_A, num_B, raw_material_cost_precalculated)
        # Action ID mapping:
        # 0: Produce_2A_0B
        # 1: Produce_1A_2B
        # 2: Produce_0A_5B
        # 3: Produce_3A_0B
        # 4: Do_Nothing

class Action(Enum):
    """
    ACTION SPACE ENUMERATION
    =======================

    Defines all possible production decisions the supplier can make.
    Each action specifies how many units of Product A and B to produce.

    FORMAT: PRODUCE_XA_YB means produce X units of A and Y units of B

    RESOURCE REQUIREMENTS:
    - Product A costs 2 raw materials per unit
    - Product B costs 1 raw material per unit
    - Total raw materials available: 10 units per day

    ACTION VALIDATION:
    - PRODUCE_2A_0B: 2*2 + 0*1 = 4 RM (valid)
    - PRODUCE_1A_2B: 1*2 + 2*1 = 4 RM (valid)
    - PRODUCE_0A_5B: 0*2 + 5*1 = 5 RM (valid)
    - PRODUCE_3A_0B: 3*2 + 0*1 = 6 RM (valid)
    - DO_NOTHING: 0*2 + 0*1 = 0 RM (valid)
    """
    PRODUCE_2A_0B = (2, 0)  # 2 units A, 0 units B (4 RM)
    PRODUCE_1A_2B = (1, 2)  # 1 unit A, 2 units B (4 RM)
    PRODUCE_0A_5B = (0, 5)  # 0 units A, 5 units B (5 RM)
    PRODUCE_3A_0B = (3, 0)  # 3 units A, 0 units B (6 RM)
    DO_NOTHING = (0, 0)     # 0 units A, 0 units B (0 RM)
```

3.4 1.4 State Space Representation

The **State** dataclass provides a complete description of the system state at any point in time. This is fundamental to our ***Markov Decision Process*** (MDP) formulation, as it contains all information necessary for optimal decision-making.

3.4.1 State Components:

1. day (int: 1-5)

- **Purpose**: Tracks progression through the 5-day episode
- **Importance**: Affects optimal strategy due to finite horizon
- **Impact**: Later days may justify more aggressive strategies

2. raw_material (int: 0-10)

- **Purpose**: Represents available production capacity
- **Constraint**: Limits feasible actions on any given day
- **Reset Mechanism**: Returns to 10 at start of each new day

3. market_state (MarketState enum)

- **Purpose**: Captures current market pricing conditions
- **Stochasticity**: Changes randomly between days
- **Strategic Impact**: Determines relative profitability of products

3.4.2 State Space Properties:

- **Size Calculation**: 5 days × 11 RM levels × 2 market states = **110 total states**
- **Tractability**: Small enough for exact dynamic programming solution
- **Completeness**: Contains all information needed for optimal decisions
- **Markovian**: Future depends only on current state, not history

3.4.3 Technical Implementation:

- **@dataclass**: Automatically generates `__init__`, `__repr__`, and `__hash__` methods
- **frozen=True**: Makes instances immutable (required for dictionary keys)
- **Type Safety**: All components have explicit type annotations

[51]:
```python
# Define state space dimensions
        # Current Day: 1 to num_days
        # Current Raw Material: 0 to initial_raw_material
        # Current Market State: 1 or 2

@dataclass(frozen=True)
class State:
    """
    STATE REPRESENTATION
    ====================

    Complete description of the system state at any point in time.
    Uses @dataclass for automatic __init__, __repr__, and __hash__ methods.
    frozen=True makes instances immutable (required for dictionary keys).

    STATE COMPONENTS:
    ----------------
    day: Current day (1-5)
         - Important for finite horizon planning
         - Affects remaining opportunities for profit

    raw_material: Available raw materials (0-10)
                 - Constrains available actions
                 - Resets to 10 at start of each day

    market_state: Current market condition (MarketState enum)
                 - Determines product prices
                 - Changes randomly each day

    STATE SPACE SIZE:
    ----------------
    Total states = Days * Raw Materials * Market States
                 = 5 * 11 * 2 = 110 states

    This manageable state space allows exact dynamic programming solutions.
    """
    day: int                # Current day (1-5)
    raw_material: int       # Available raw materials (0-10)
    market_state: MarketState  # Current market condition
```

3.5 1.5 Smart Supplier Environment Implementation

The **SmartSupplierEnvironment** class is the core of our MDP implementation. It defines the complete dynamics of the Smart Supplier problem, including state transitions, reward calculations, and constraint enforcement.

3.5.1 Environment Characteristics:

Finite Horizon Problem

- **Episode Length**: Exactly 5 days per episode
- **Termination**: Natural endpoint after day 5
- **Implication**: No infinite horizon considerations needed

Stochastic Transitions

- **Market Changes**: Random market state transitions each day
- **Probability Distribution**: 50% chance for each market state
- **Independence**: Market state transitions are memoryless

Deterministic Rewards

- **Predictability**: Given state and action, reward is always the same
- **Calculation**: Revenue = units_produced × current_market_price
- **Simplicity**: No reward uncertainty, only transition uncertainty

Resource Constraints

- **Daily Limit**: 10 raw materials available each day
- **Reset Mechanism**: Resources replenish to full each morning
- **Constraint Enforcement**: Infeasible actions result in zero production

3.5.2 Key Methods Overview:

1. `is_action_feasible()`: Validates resource constraints
2. `get_reward()`: Calculates immediate profit from production
3. `get_next_states()`: Returns possible transitions with probabilities
4. `get_all_states()`: Enumerates complete state space for DP
5. `get_feasible_actions()`: Identifies legal actions per state

3.5.3 MDP Formulation:

- **States (S)**: {day, raw_material, market_state} combinations
- **Actions (A)**: Production decisions from Action enum
- **Transitions (P)**: Stochastic market changes, deterministic day progression
- **Rewards (R)**: Profit from selling produced units
- **Discount ( )**: 1.0 (no discounting for finite horizon)

[52]:
```
class SmartSupplierEnvironment:
    """
    SMART SUPPLIER ENVIRONMENT IMPLEMENTATION
    ========================================

    This class implements the complete environment dynamics for the Smart␣
↪Supplier
    problem, including state transitions, reward calculations, and constraint␣
↪checking.

    ENVIRONMENT CHARACTERISTICS:
    ---------------------------
    - FINITE HORIZON: Episodes last exactly 5 days
    - STOCHASTIC TRANSITIONS: Market state changes randomly
    - DETERMINISTIC REWARDS: Given state and action, reward is deterministic
    - RESOURCE CONSTRAINTS: Limited raw materials constrain feasible actions
    - DAILY RESET: Raw materials reset to 10 each day

    TRANSITION DYNAMICS:
    -------------------
    When an action is taken in state (day, rm, market):
    1. Check if action is feasible given raw materials
    2. Calculate immediate reward from production
    3. Advance to next day with reset raw materials
    4. Market state transitions with 50% probability each

    REWARD STRUCTURE:
```

```
    ----------------
    Rewards are calculated as: (units_A * price_A) + (units_B * price_B)
    Prices depend on current market state:
    - Market State 1: A=$8, B=$2
    - Market State 2: A=$3, B=$5
    """

    def __init__(self):
        """
        ENVIRONMENT INITIALIZATION
        =========================

        Set up all environment parameters including:
        - Product costs and prices
        - Raw material limits
        - Episode duration
        - Market transition probabilities
        """
        # PRODUCTION COSTS (in raw materials)
        self.cost_A = 2  # Product A costs 2 RM per unit
        self.cost_B = 1  # Product B costs 1 RM per unit

        # DAILY RESOURCE ALLOCATION
        self.initial_raw_material = 10  # Start each day with 10 RM

        # EPISODE CONFIGURATION
        self.max_days = 5  # Episode lasts 5 days

        # MARKET PRICING STRUCTURE
        # Dictionary mapping market state to (price_A, price_B)

        # Structure: {Market_State_ID: {'A_price': X, 'B_price': Y}}

        self.market_prices = {
            MarketState.HIGH_DEMAND_A: (8, 2),  # A favored: A=$8, B=$2
            MarketState.HIGH_DEMAND_B: (3, 5)   # B favored: A=$3, B=$5
        }

        # MARKET TRANSITION PROBABILITIES
        # 50% chance of each market state each day (independent)
        self.market_transition_prob = 0.5

        print("SMART SUPPLIER ENVIRONMENT INITIALIZED")
        print("=" * 50)
        print(f"Episode length: {self.max_days} days")
        print(f"Daily raw materials: {self.initial_raw_material} units")
        print(f"Product A cost: {self.cost_A} RM per unit")
        print(f"Product B cost: {self.cost_B} RM per unit")
        print("\nMarket Pricing:")
        for market, (price_a, price_b) in self.market_prices.items():
            print(f"  {market.name}: A=${price_a}, B=${price_b}")

    def is_action_feasible(self, state: State, action: Action) -> bool:
        """
        ACTION FEASIBILITY CHECK
        ========================

        Determines whether a given action can be executed in the current state.
        An action is feasible if the required raw materials don't exceed available resources.

        FEASIBILITY CONSTRAINT:
        -----------------------
        total_cost = (units_A * cost_A) + (units_B * cost_B)   available_RM

        INFEASIBLE ACTION HANDLING:
        ---------------------------
        If an action is infeasible, it results in no production and zero reward.
        This models the real-world constraint that you can't produce more than
        your resources allow.

        Args:
            state: Current state of the environment
            action: Action to check for feasibility

        Returns:
            bool: True if action can be executed, False otherwise
        """
        units_A, units_B = action.value
        total_cost = (units_A * self.cost_A) + (units_B * self.cost_B)
        return total_cost <= state.raw_material

    # get reward function
    def get_reward(self, state: State, action: Action) -> float:
        """
        REWARD CALCULATION
        ==================

        Calculates the immediate reward for taking a specific action in a given state.
        Reward represents the profit from selling produced units at current market prices.

        REWARD FORMULA:
```

```
        --------------
        If action is feasible:
            reward = (units_A * current_price_A) + (units_B * current_price_B)
        If action is infeasible:
            reward = 0 (no production occurs)

        MARKET PRICE DEPENDENCY:
        -----------------------
        Prices depend on current market state:
        - HIGH_DEMAND_A: Favors Product A (A=$8, B=$2)
        - HIGH_DEMAND_B: Favors Product B (A=$3, B=$5)

        ECONOMIC INTERPRETATION:
        -----------------------
        This reward structure captures the core economic trade-off:
        - Different products have different profitability in different markets
        - Resource constraints limit production possibilities
        - Optimal decisions must consider both current prices and resource costs

        Args:
            state: Current state (includes market condition)
            action: Production decision to evaluate

        Returns:
            float: Immediate profit from the action
        """
        # Check if action is feasible given resource constraints
        if not self.is_action_feasible(state, action):
            return 0.0  # No production, no profit

        # Extract production quantities from action
        units_A, units_B = action.value

        # Get current market prices
        price_A, price_B = self.market_prices[state.market_state]

        # Calculate total revenue (profit)
        revenue = (units_A * price_A) + (units_B * price_B)

        return float(revenue)

    def get_next_states(self, state: State, action: Action) ->␣
↪List[Tuple[State, float]]:
        """
        STATE TRANSITION DYNAMICS
        ========================
        Returns all possible next states and their probabilities after taking␣
↪an action.
        This is crucial for dynamic programming algorithms that need to␣
↪consider all
        possible outcomes when computing expected values.

        TRANSITION MECHANICS:
        --------------------
        1. Day advances by 1 (deterministic)
        2. Raw materials reset to 10 (deterministic)
        3. Market state changes (stochastic)

        MARKET STATE TRANSITIONS:
        ------------------------
        Each day, market state is independently determined:
        - P(HIGH_DEMAND_A) = 0.5
        - P(HIGH_DEMAND_B) = 0.5

        This creates a Markov process where future market conditions don't␣
↪depend
        on past market conditions (realistic assumption for daily price␣
↪fluctuations).

        EPISODE TERMINATION:
        -------------------
        If we're on the last day (day 5), episode terminates and no next states␣
↪exist.

        Args:
            state: Current state
            action: Action taken (doesn't affect transitions in this␣
↪environment)

        Returns:
            List of (next_state, probability) tuples
        """
        # Check for episode termination
        if state.day >= self.max_days:
            return []  # No next states - episode ends

        # Calculate next day
        next_day = state.day + 1

        # Raw materials reset to full amount each day
        next_raw_material = self.initial_raw_material
        # Generate all possible next states (one for each market condition)
        next_states = []

        for market_state in MarketState:
            next_state = State(
                day=next_day,
                raw_material=next_raw_material,
                market_state=market_state
            )
            # Each market state has equal probability (50%)
            probability = self.market_transition_prob
            next_states.append((next_state, probability))

        return next_states

    def get_all_states(self) -> List[State]:
        """
        COMPLETE STATE SPACE ENUMERATION
        ===============================

        Generates all possible states in the environment for exact dynamic␣
↪programming.
        This is feasible because our state space is relatively small (110␣
↪states).

        STATE SPACE STRUCTURE:
        ---------------------
        - Days: 1, 2, 3, 4, 5 (5 values)
        - Raw Materials: 0, 1, 2, ..., 10 (11 values)
        - Market States: HIGH_DEMAND_A, HIGH_DEMAND_B (2 values)

        Total: 5 × 11 × 2 = 110 states

        DYNAMIC PROGRAMMING REQUIREMENT:
        -------------------------------
        Exact DP algorithms require knowing all states to:
        1. Initialize value function for all states
        2. Perform value updates across all states
        3. Extract optimal policy for all states

        Returns:
            List of all possible State objects
        """
        states = []

        # Enumerate all combinations of state variables
        for day in range(1, self.max_days + 1):
            for raw_material in range(self.initial_raw_material + 1):
                for market_state in MarketState:
                    state = State(
                        day=day,
                        raw_material=raw_material,
                        market_state=market_state
                    )
                    states.append(state)

        return states

    def get_feasible_actions(self, state: State) -> List[Action]:
        """
        FEASIBLE ACTION IDENTIFICATION
        ==============================

        Returns all actions that can be legally executed in the given state.
        This is essential for policy optimization - we only consider actions
        that don't violate resource constraints.

        CONSTRAINT CHECKING:
        -------------------
        For each action, verify that:
        required_RM = (units_A × cost_A) + (units_B × cost_B)   available_RM

        EMPTY ACTION SET HANDLING:
        -------------------------
        In extreme cases (very low raw materials), only DO_NOTHING might be␣
↪feasible.
        This ensures the agent always has at least one legal action.

        Args:
            state: Current state to check actions against

        Returns:
            List of feasible Action enums
        """
        feasible_actions = []

        for action in Action:
            if self.is_action_feasible(state, action):
                feasible_actions.append(action)

        return feasible_actions
```

4 2. Dynamic Programming Implementation (Value Iteration or Policy Iteration)

4.1 2.1 Value Iteration Algorithm - Theoretical Foundation

Value Iteration is a fundamental dynamic programming algorithm that computes the optimal value function V*(s) and policy* (s) for Markov Decision Processes. It's particularly well-suited for our Smart Supplier problem due to the finite horizon and manageable state space.

4.1.1 Mathematical Foundation:

Bellman Optimality Equation The core principle behind Value Iteration is the Bellman optimality equation:

```
V*(s) = max_a Σ_{s'} P(s'|s,a) × [R(s,a,s') +   × V*(s')]
```

Where: - **V*(s)**: Optimal value (expected total reward) from state s - **max_a**: Optimization over all available actions - **P(s'|s,a)**: Transition probability to next state s' - **R(s,a,s')**: Immediate reward from taking action a in state s - : Discount factor (1.0 for our finite horizon problem)

Algorithm Steps

1. **Initialize**: V (s) = 0 for all states s
2. **Iterate**: For k = 0, 1, 2, … until convergence:

   ```
   V_{k+1}(s) = max_a Σ_{s'} P(s'|s,a) × [R(s,a,s') +   × V_k(s')]
   ```

3. **Extract Policy**: *(s) = argmax_a Σ_{s'} P(s'|s,a) × [R(s,a,s') + × V*(s')]

Convergence Properties

- **Guaranteed Convergence**: For finite MDPs with proper conditions
- **Contraction Mapping**: Each iteration brings us closer to optimal solution
- **Linear Convergence**: Exponential improvement in accuracy

4.1.2 Why Value Iteration for Smart Supplier?

1. **Finite State Space**: Only 110 states makes exact computation feasible
2. **Finite Horizon**: 5-day episodes don't require infinite horizon considerations
3. **Optimal Solution**: Provides globally optimal policy (not just locally optimal)
4. **Computational Efficiency**: Faster than policy iteration for this problem size
5. **Theoretical Guarantees**: Proven convergence to optimal solution

[53]:
```python
# Value Iteration function
class ValueIteration:
    """
    VALUE ITERATION ALGORITHM IMPLEMENTATION
    ========================================

    Value Iteration is a dynamic programming algorithm that computes the optimal
    value function V*(s) and optimal policy *(s) for Markov Decision Processes.
```

```
    ALGORITHM OVERVIEW:
    ==================
    Iteratively update value estimates using the Bellman optimality equation␣
↪until convergence:

    V_{k+1}(s) = max_a Σ_{s'} P(s'|s,a) * [R(s,a,s') +   * V_k(s')]

    ALGORITHM STEPS:
    ===============
    1. Initialize V(s) = 0 for all states s
    2. Repeat until convergence:
       a. For each state s:
          - For each action a:
            - Calculate expected value: Q(s,a) = Σ P(s'|s,a)[R(s,a,s') +  V(s')]
          - Update V(s) = max_a Q(s,a)
    3. Extract optimal policy:  *(s) = argmax_a Q(s,a)

    CONVERGENCE CRITERIA:
    ====================
    Stop if maximum change in value function across all states falls below a␣
↪threshold (typically 1e-6)

    FINITE HORIZON CONSIDERATIONS:
    =============================
    For this 5-day problem, we use  =1.0 (no discounting) since:
    - Episode has definite end point
    - All rewards are equally important regardless of timing
    - Terminal states (day > 5) have V = 0

    COMPUTATIONAL COMPLEXITY:
    ========================
    - Time: O(iterations * |S| * |A| * |S|)
    - Space: O(|S|)
    - For our problem: ~O(iterations * 110 * 5 * 110) operations
    """

    def __init__(self, environment: SmartSupplierEnvironment, gamma: float = 1.
↪0,
                 tolerance: float = 1e-6, max_iterations: int = 1000):
        """
        VALUE ITERATION INITIALIZATION
        =============================

        Args:
            environment: The Smart Supplier environment
            gamma: Discount factor (1.0 for finite horizon problems)
            tolerance: Convergence threshold for value function changes
            max_iterations: Maximum iterations to prevent infinite loops
        """
        self.env = environment
        self.gamma = gamma
        self.tolerance = tolerance
        self.max_iterations = max_iterations

        # Initialize data structures
        self.states = self.env.get_all_states()
        self.value_function = {}  # V(s) for each state
        self.policy = {}          #  (s) for each state
        self.q_values = {}        # Q(s,a) for state-action pairs

        # Performance tracking
        self.iteration_count = 0
        self.convergence_history = []

        print("VALUE ITERATION ALGORITHM INITIALIZED")
        print("=" * 40)
        print(f"State space size: {len(self.states)}")
        print(f"Discount factor: {self.gamma}")
        print(f"Convergence tolerance: {self.tolerance}")
        print(f"Maximum iterations: {self.max_iterations}")

    def initialize_value_function(self):
        """
        VALUE FUNCTION INITIALIZATION
        ============================

        Initialize the value function for all states. For finite horizon␣
↪problems,
        we typically start with V(s) = 0 for all states.

        TERMINAL STATE HANDLING:
        -----------------------
        States beyond the episode horizon (day > max_days) have V = 0␣
↪permanently,
        representing that no further rewards can be obtained.

        INITIAL VALUE CHOICE:
        --------------------
        V(s) = 0 is a common choice because:
        - Conservative estimate (underestimates true values initially)
        - Mathematically sound (values will increase toward optimal)
        - Simple and interpretable
        """
        print("\nInitializing value function...")

        for state in self.states:
            # All states start with zero value
            self.value_function[state] = 0.0

        print(f"Initialized {len(self.value_function)} states with V(s) = 0")

    def compute_q_value(self, state: State, action: Action) -> float:
        """
        Q-VALUE COMPUTATION (ACTION-VALUE FUNCTION)
        ==========================================

        Computes Q(s,a) = expected total reward from taking action a in state s
        and then following the optimal policy thereafter.

        Q-VALUE FORMULA:
        ---------------
        Q(s,a) = R(s,a) +   × Σ_{s'} P(s'|s,a) × V(s')

        Where:
        - R(s,a): Immediate reward from taking action a in state s
        -  : Discount factor
        - P(s'|s,a): Probability of transitioning to state s'
        - V(s'): Current value estimate for next state s'

        EXPECTATION CALCULATION:
        -----------------------
        Since our environment is stochastic (random market transitions), we must
        compute the expected value over all possible next states weighted by
        their transition probabilities.

        Args:
            state: Current state
            action: Action to evaluate

        Returns:
            float: Q-value for the state-action pair
        """
        # Get immediate reward
        immediate_reward = self.env.get_reward(state, action)

        # Get all possible next states and their probabilities
        next_states = self.env.get_next_states(state, action)

        # Compute expected future value
        expected_future_value = 0.0
        for next_state, probability in next_states:
            expected_future_value += probability * self.
↪value_function[next_state]

        # Q-value = immediate reward + discounted expected future value
        q_value = immediate_reward + self.gamma * expected_future_value

        return q_value

    def value_iteration_step(self) -> float:
        """
        SINGLE VALUE ITERATION UPDATE
        ============================

        Performs one complete sweep through all states, updating their values
        according to the Bellman optimality equation.

        BELLMAN OPTIMALITY UPDATE:
        --------------------------
        For each state s:
        V_{new}(s) = max_a Q(s,a)

        This represents the maximum expected total reward achievable from state␣
↪s.

        CONVERGENCE MEASUREMENT:
        -----------------------
        We track the maximum absolute change in value function:
        max_change = max_s |V_{new}(s) - V_{old}(s)|

        When max_change < tolerance, the algorithm has converged.

        SYNCHRONOUS UPDATES:
        -------------------
        We compute all new values before updating any state's value.
        This ensures consistent computation across the entire state space.

        Returns:
            float: Maximum absolute change in value function
        """
        new_values = {}
        max_change = 0.0

        # Compute new values for all states
        for state in self.states:
            # Get all feasible actions for this state
            feasible_actions = self.env.get_feasible_actions(state)
            if not feasible_actions:
                # Edge case: no feasible actions (shouldn't happen in our␣
↪environment)
                new_values[state] = 0.0
                continue

            # Compute Q-values for all feasible actions
            q_values = []
            for action in feasible_actions:
                q_val = self.compute_q_value(state, action)
                q_values.append(q_val)

            # Bellman optimality: V(s) = max_a Q(s,a)
            new_values[state] = max(q_values)

            # Track convergence
            change = abs(new_values[state] - self.value_function[state])
            max_change = max(max_change, change)

        # Update value function (synchronous update)
        self.value_function = new_values

        return max_change

    def extract_policy(self):
        """
        OPTIMAL POLICY EXTRACTION
        ========================

        After value iteration converges, extract the optimal policy by selecting
        the action that maximizes Q-value in each state.

        POLICY EXTRACTION FORMULA:
        -------------------------
         *(s) = argmax_a Q(s,a)

        This gives us the optimal action to take in each state to maximize
        expected total reward.

        Q-VALUE STORAGE:
        ---------------
        We also store Q-values for analysis and debugging purposes.
        Q-values provide insight into the relative value of different actions.
        """
        print("\nExtracting optimal policy...")
        for state in self.states:
            feasible_actions = self.env.get_feasible_actions(state)

            if not feasible_actions:
                # Edge case handling
                self.policy[state] = Action.DO_NOTHING
                continue

            # Compute Q-values for all feasible actions
            best_action = None
            best_q_value = float('-inf')
            state_q_values = {}

            for action in feasible_actions:
                q_val = self.compute_q_value(state, action)
                state_q_values[action] = q_val

                if q_val > best_q_value:
                    best_q_value = q_val
                    best_action = action

            # Store optimal action and Q-values
            self.policy[state] = best_action
            self.q_values[state] = state_q_values

        print(f"Extracted policy for {len(self.policy)} states")

    def solve(self) -> Tuple[Dict[State, float], Dict[State, Action]]:
        """
        COMPLETE VALUE ITERATION ALGORITHM
        =================================

        Runs the full value iteration algorithm until convergence, then extracts
        the optimal policy.

        ALGORITHM EXECUTION:
        -------------------
        1. Initialize value function
        2. Iterate until convergence:
           - Update all state values using Bellman equation
           - Check for convergence
        3. Extract optimal policy from converged value function

        CONVERGENCE MONITORING:
        ----------------------
        We track convergence history to analyze algorithm behavior and
        ensure proper convergence.
        Returns:
            Tuple of (value_function, optimal_policy)
        """
        print("\n" + "="*60)
        print("STARTING VALUE ITERATION ALGORITHM")
        print("="*60)

        # Step 1: Initialize
        self.initialize_value_function()

        # Step 2: Iterate until convergence
        print("\nIterating toward optimal value function...")
        for iteration in range(self.max_iterations):
            max_change = self.value_iteration_step()
            self.convergence_history.append(max_change)
            self.iteration_count = iteration + 1

            # Progress reporting
            if iteration % 10 == 0 or max_change < self.tolerance:
                print(f"  Iteration {iteration+1:3d}: Max change = {max_change:.
↪8f}")

            # Check for convergence
            if max_change < self.tolerance:
                print(f"\n CONVERGED after {iteration+1} iterations!")
                print(f"  Final max change: {max_change:.2e}")
                break
        else:
            print(f"\n Reached maximum iterations ({self.max_iterations})")
            print(f"  Final max change: {max_change:.2e}")

        # Step 3: Extract optimal policy
        self.extract_policy()

        print(f"\nVALUE ITERATION COMPLETE")
        print(f"   Convergence achieved in {self.iteration_count} iterations")

        return self.value_function, self.policy
```

5 3. Simulation and Policy Analysis

5.1 3.1 Policy Simulation and Performance Evaluation

After computing the optimal policy using Value Iteration, we need to validate its performance through **Monte Carlo simulation**. The **PolicySimulator** class provides comprehensive evaluation of the learned policy's real-world performance.

5.1.1 Simulation Methodology:

Multi-Episode Testing

- **Sample Size**: Run 1000+ independent episodes for statistical significance
- **Episode Structure**: Each episode follows the 5-day Smart Supplier scenario
- **Stochastic Elements**: Market states change randomly as in real environment
- **Policy Execution**: Follow optimal policy decisions at each step

Performance Metrics

1. **Expected Profit**: Average total reward across all episodes
2. **Risk Assessment**: Standard deviation of profits (volatility measure)
3. **Confidence Intervals**: Statistical bounds on expected performance
4. **Action Distribution**: Frequency analysis of different production decisions
5. **Consistency Check**: Verify policy behaves rationally across scenarios

5.1.2 Validation Objectives:

Theoretical vs Practical

- **Value Function Validation**: Simulated performance should match theoretical V*(s)
- **Policy Consistency**: Decisions should adapt correctly to state changes
- **Constraint Compliance**: All actions must respect resource limitations

Strategy Analysis

- **Market Adaptation**: Does policy favor Product A in Market State 1?
- **Resource Management**: How does policy handle low raw material situations?
- **Time Sensitivity**: Does strategy change as episode progresses?

5.1.3 Statistical Rigor:

- **Large Sample Size**: 1000 episodes provide robust statistics
- **Confidence Intervals**: 95% confidence bounds on mean performance
- **Distribution Analysis**: Understanding profit variability and risk
- **Comparative Benchmarking**: Performance relative to simple heuristics

[54]:
```
# simulate policy function - Simulates the learned policy over multiple runs to␣
↪evaluate performance
class PolicySimulator:
    """
    POLICY SIMULATION AND PERFORMANCE EVALUATION
    ============================================

    This class simulates the learned optimal policy over multiple episodes
    to evaluate its real-world performance and validate the theoretical
    optimal value function.

    SIMULATION METHODOLOGY:
```

```
    ======================
    1. Run multiple independent episodes (e.g., 1000 runs)
    2. In each episode, follow the optimal policy for 5 days
    3. Record total rewards and decision patterns
    4. Compute performance statistics

    VALIDATION CHECKS:
    =================
    - Average simulated reward should match theoretical value function
    - Policy should adapt correctly to market state changes
    - Resource constraints should be respected in all decisions

    PERFORMANCE METRICS:
    ===================
    - Average total profit per episode
    - Standard deviation of profits (risk measure)
    - Action frequency distribution
    - Market adaptation effectiveness
    """

    def __init__(self, environment: SmartSupplierEnvironment,
                 policy: Dict[State, Action]):
        """
        Initialize the policy simulator.

        Args:
            environment: The Smart Supplier environment
            policy: Optimal policy to simulate
        """
        self.env = environment
        self.policy = policy
        self.simulation_results = []

    def simulate_episode(self) -> Tuple[float, List[Tuple[State, Action,␣
↪float]]]:
        """
        SINGLE EPISODE SIMULATION
        ========================

        Simulates one complete 5-day episode following the optimal policy.

        EPISODE FLOW:
        ------------
        1. Start on Day 1 with 10 RM and random market state
        2. For each day:
           a. Observe current state
           b. Take action according to optimal policy
           c. Receive reward
           d. Transition to next day (reset RM, new market state)
        3. Episode ends after Day 5

        Returns:
            Tuple of (total_reward, episode_history)
        """
        # Initialize episode
        total_reward = 0.0
        episode_history = []

        # Random initial market state
        initial_market = random.choice(list(MarketState))
        current_state = State(
            day=1,
            raw_material=self.env.initial_raw_material,
            market_state=initial_market
        )

        # Simulate 5 days
        for day in range(5):
            # Get optimal action for current state
            action = self.policy[current_state]

            # Calculate reward
            reward = self.env.get_reward(current_state, action)
            total_reward += reward

            # Record step
            episode_history.append((current_state, action, reward))

            # Transition to next day (if not last day)
            if day < 4:  # Days 0-3 transition to next day
                next_market = random.choice(list(MarketState))
                current_state = State(
                    day=current_state.day + 1,
                    raw_material=self.env.initial_raw_material,
                    market_state=next_market
                )

        return total_reward, episode_history

    def run_simulation(self, num_episodes: int = 1000) -> Dict:
        """
        COMPREHENSIVE POLICY SIMULATION
        ==============================
        Runs multiple episodes to evaluate policy performance statistically.

        STATISTICAL ANALYSIS:
        ---------------------
        - Mean performance: Expected profit under optimal policy
        - Variance analysis: Risk and consistency of performance
        - Confidence intervals: Statistical reliability bounds
        - Action pattern analysis: Behavioral consistency

        Args:
            num_episodes: Number of episodes to simulate

        Returns:
            Dictionary containing simulation results and statistics
        """
        print(f"\n" + "="*70)
        print(f"POLICY SIMULATION: {num_episodes} EPISODES")
        print("="*70)

        self.simulation_results = []
        episode_rewards = []
        all_actions = []

        # Run simulations
        for episode in range(num_episodes):
            total_reward, episode_history = self.simulate_episode()
            episode_rewards.append(total_reward)

            # Collect actions for analysis
            for state, action, reward in episode_history:
                all_actions.append(action)

            # Progress reporting
            if (episode + 1) % 100 == 0:
                current_avg = np.mean(episode_rewards)
                print(f"  Episode {episode+1:4d}: Running average =␣
↪${current_avg:.2f}")

        # Calculate statistics
        mean_reward = np.mean(episode_rewards)
        std_reward = np.std(episode_rewards)
        min_reward = np.min(episode_rewards)
        max_reward = np.max(episode_rewards)

        # Confidence interval (95%)
        confidence_margin = 1.96 * std_reward / np.sqrt(num_episodes)
        ci_lower = mean_reward - confidence_margin
        ci_upper = mean_reward + confidence_margin

        # Action frequency analysis
        action_counts = {action: 0 for action in Action}
        for action in all_actions:
            action_counts[action] += 1

        total_actions = len(all_actions)
        action_percentages = {action: (count/total_actions)*100
                              for action, count in action_counts.items()}

        # Compile results
        results = {
            'num_episodes': num_episodes,
            'episode_rewards': episode_rewards,
            'mean_reward': mean_reward,
            'std_reward': std_reward,
            'min_reward': min_reward,
            'max_reward': max_reward,
            'confidence_interval': (ci_lower, ci_upper),
            'action_counts': action_counts,
            'action_percentages': action_percentages
        }

        # Print summary
        print(f"\nSIMULATION RESULTS SUMMARY:")
        print(f"   Episodes Simulated: {num_episodes}")
        print(f"   Average Profit: ${mean_reward:.2f} ± ${std_reward:.2f}")
        print(f"   95% Confidence Interval: ${ci_lower:.2f} - ${ci_upper:.2f}")
        print(f"   Profit Range: ${min_reward:.2f} - ${max_reward:.2f}")

        print(f"\nACTION USAGE DISTRIBUTION:")
        for action, percentage in action_percentages.items():
            units_a, units_b = action.value
            print(f"   {action.name:15} | {percentage:5.1f}% | Produces:␣
↪{units_a}A, {units_b}B")

        return results
```

5.2 3.2 Comprehensive Policy Analysis Framework

The **PolicyAnalyzer** class provides in-depth analysis of the optimal policy learned through Value Iteration. This analysis is crucial for understanding how the agent makes decisions and validating that the learned strategy is economically rational.

5.2.1 Analysis Dimensions:

1. Market State Dependency Analysis

- **Research Question**: How does the optimal policy adapt to different market conditions?
- **Expected Behavior**:
  - Favor Product A when Market State = HIGH_DEMAND_A (A=$8, B=$2)
  - Favor Product B when Market State = HIGH_DEMAND_B (A=$3, B=$5)
- **Validation**: Ensure policy demonstrates market awareness and adaptation

2. **Resource Constraint Impact Analysis**

- **Research Question**: How do available raw materials influence production decisions?
- **Expected Patterns**:
  - **Low RM (0-3)**: Conservative strategies, prefer efficient Product B
  - **Medium RM (4-6)**: Balanced production strategies

  - **High RM (7-10)**: Can afford expensive Product A production
- **Insight**: Optimal resource allocation under scarcity

3. **Time Horizon Effects Analysis**

- **Research Question**: How does strategy change as episode progresses?
- **Expected Evolution**:
  - **Early Days (1-2)**: Conservative, balanced approaches
  - **Middle Days (3-4)**: Adaptive strategies based on market
  - **Final Day (5)**: Aggressive profit maximization
- **Rationale**: Finite horizon affects risk tolerance and opportunity cost

4. **Value Function Pattern Analysis**

- **Research Question**: Which states are most/least valuable?
- **Insights Sought**:
  - **High-Value States**: Early days + favorable market + full resources
  - **Low-Value States**: Late days + unfavorable market + depleted resources
  - **Value Gradients**: How value changes across state dimensions

5.2.2 Economic Validation:

- **Rational Behavior**: Decisions should maximize expected profit
- **Market Efficiency**: Policy should exploit price differentials
- **Resource Optimization**: Efficient allocation of scarce raw materials
- **Strategic Coherence**: Consistent decision patterns across similar states

```
[ ]: # analyze policy function - Analyzes and prints snippets of the learned optimal␣
     ↪policy
     class PolicyAnalyzer:
         """
         OPTIMAL POLICY ANALYSIS AND VISUALIZATION
         ========================================

         This class provides comprehensive analysis of the learned optimal policy,
         examining how decisions change based on different state variables.
```

```
    ANALYSIS DIMENSIONS:
    ===================
    1. MARKET STATE DEPENDENCY: How does policy change with market conditions?
    2. RESOURCE CONSTRAINT IMPACT: How do limited raw materials affect␣
↪decisions?
    3. TIME HORIZON EFFECTS: How does proximity to episode end influence␣
↪strategy?
    4. VALUE FUNCTION ANALYSIS: What are the most/least valuable states?

    INSIGHTS PROVIDED:
    =================
    - Strategic patterns in optimal decision making
    - Trade-offs between different products under different conditions
    - Resource management strategies
    - Time-sensitive decision patterns
    """

    def __init__(self, environment: SmartSupplierEnvironment,
                 value_function: Dict[State, float],
                 policy: Dict[State, Action]):
        """
        Initialize the policy analyzer.

        Args:
            environment: The Smart Supplier environment
            value_function: Optimal value function from Value Iteration
            policy: Optimal policy from Value Iteration
        """
        self.env = environment
        self.value_function = value_function
        self.policy = policy

    def analyze_market_dependency(self):
        """
        MARKET STATE DEPENDENCY ANALYSIS
        ===============================

        Analyzes how the optimal policy changes based on market conditions.
        This reveals whether the agent successfully learns to adapt to
        changing market prices.

        KEY QUESTIONS:
        -------------
        - Does the policy favor Product A when Market State 1 (high A demand)?
        - Does the policy favor Product B when Market State 2 (high B demand)?
        - Are there states where the policy is invariant to market conditions?
        """
        print("\n" + "="*70)
        print("MARKET STATE DEPENDENCY ANALYSIS")
        print("="*70)

        # Analyze policy for each market state
        for market_state in MarketState:
            print(f"\n OPTIMAL STRATEGY IN {market_state.name}:")
            print(f"   Market Prices: A=${self.env.
↪market_prices[market_state][0]}, "
                  f"B=${self.env.market_prices[market_state][1]}")

            # Count action frequencies for this market state
            action_counts = {action: 0 for action in Action}
            total_states = 0

            for state in self.value_function.keys():
                if state.market_state == market_state:
                    action = self.policy[state]
                    action_counts[action] += 1
                    total_states += 1

            # Display action preferences
            print(f"   Action Distribution ({total_states} states):")
            for action, count in action_counts.items():
                percentage = (count / total_states) * 100 if total_states > 0␣
↪else 0
                units_a, units_b = action.value
                print(f"     {action.name:15} | {count:2d} states ({percentage:
↪5.1f}%) | "
                      f"Produces: {units_a}A, {units_b}B")

    def analyze_resource_dependency(self):
        """
        RESOURCE CONSTRAINT IMPACT ANALYSIS
        ===================================

        Examines how available raw materials influence optimal decisions.
        This shows whether the agent learns efficient resource management.

        EXPECTED PATTERNS:
        ------------------
        - Low RM: Prefer cheaper Product B or do nothing
        - High RM: Can afford expensive Product A production
        - Medium RM: Balanced production strategies
        """
        print("\n" + "="*70)
        print("RESOURCE CONSTRAINT IMPACT ANALYSIS")
        print("="*70)

        # Group states by raw material level
        rm_levels = [0, 2, 4, 6, 8, 10]  # Sample key resource levels

        for rm_level in rm_levels:
            print(f"\nSTRATEGY WITH {rm_level} RAW MATERIALS:")

            # Find states with this resource level
            relevant_states = [s for s in self.value_function.keys()
                             if s.raw_material == rm_level]

            if not relevant_states:
                print("   No states with this resource level")
                continue

            # Analyze action distribution
            action_counts = {action: 0 for action in Action}
            for state in relevant_states:
                action = self.policy[state]
                action_counts[action] += 1

            total_states = len(relevant_states)
            print(f"   Action Distribution ({total_states} states):")

            for action, count in action_counts.items():
                if count > 0:
                    percentage = (count / total_states) * 100
                    units_a, units_b = action.value
                    cost = units_a * 2 + units_b * 1
                    feasible = " " if cost <= rm_level else " "
                    print(f"     {action.name:15} | {count:2d} states␣
↪({percentage:5.1f}%) | "
                          f"Cost: {cost}RM {feasible}")

    def analyze_time_dependency(self):
        """
        TIME HORIZON EFFECTS ANALYSIS
        ============================

        Studies how the optimal policy changes as the episode progresses.
        This reveals time-sensitive strategic adaptations.

        EXPECTED PATTERNS:
        -----------------
        - Early days: More conservative, balanced strategies
        - Later days: More aggressive, exploit remaining opportunities
        - Last day: Maximum exploitation regardless of future consequences
        """
        print("\n" + "="*70)
        print("TIME HORIZON EFFECTS ANALYSIS")
        print("="*70)

        for day in range(1, self.env.max_days + 1):
            print(f"\n  DAY {day} STRATEGY:")

            # Find states for this day
            day_states = [s for s in self.value_function.keys() if s.day == day]

            # Analyze action distribution
            action_counts = {action: 0 for action in Action}
            total_value = 0.0

            for state in day_states:
                action = self.policy[state]
                action_counts[action] += 1
                total_value += self.value_function[state]

            total_states = len(day_states)
            avg_value = total_value / total_states if total_states > 0 else 0

            print(f"   Average State Value: ${avg_value:.2f}")
            print(f"   Action Distribution ({total_states} states):")

            for action, count in action_counts.items():
                if count > 0:
                    percentage = (count / total_states) * 100
                    units_a, units_b = action.value
                    print(f"     {action.name:15} | {count:2d} states␣
↪({percentage:5.1f}%) | "
                          f"Produces: {units_a}A, {units_b}B")

    def analyze_value_function(self):
        """
        VALUE FUNCTION ANALYSIS
        ======================

        Examines the learned value function to identify:
        - Most valuable states (best situations to be in)
        - Least valuable states (situations to avoid)
        - Value patterns across different state dimensions
        """
        print("\n" + "="*70)
        print("VALUE FUNCTION ANALYSIS")
        print("="*70)

        # Find extreme value states
        states_values = [(state, value) for state, value in self.value_function.
↪items()]
        states_values.sort(key=lambda x: x[1], reverse=True)

        print("\nTOP 10 MOST VALUABLE STATES:")
        for i, (state, value) in enumerate(states_values[:10]):
            action = self.policy[state]
            units_a, units_b = action.value
            print(f"   {i+1:2d}. Day {state.day}, RM={state.raw_material:2d}, "
                  f"{state.market_state.name:15} | V=${value:6.2f} | "
                  f"Action: {units_a}A,{units_b}B")

        print("\nBOTTOM 10 LEAST VALUABLE STATES:")
        for i, (state, value) in enumerate(states_values[-10:]):
            action = self.policy[state]
            units_a, units_b = action.value
            print(f"   {i+1:2d}. Day {state.day}, RM={state.raw_material:2d}, "
                  f"{state.market_state.name:15} | V=${value:6.2f} | "
                  f"Action: {units_a}A,{units_b}B")

        # Calculate statistics
        values = [v for _, v in states_values]
        print(f"\nVALUE FUNCTION STATISTICS:")
        print(f"   Maximum Value: ${max(values):.2f}")
        print(f"   Minimum Value: ${min(values):.2f}")
        print(f"   Average Value: ${sum(values)/len(values):.2f}")

        # Calculate standard deviation manually
        mean_val = sum(values) / len(values)
        variance = sum((x - mean_val) ** 2 for x in values) / len(values)
        std_dev = variance ** 0.5
        print(f"   Standard Deviation: ${std_dev:.2f}")
```

6 4. Impact of Dynamics Analysis

6.1 4.1 Impact of Market Dynamics - Theoretical Framework

Understanding how **market uncertainty** affects optimal decision-making is crucial for validating our dynamic programming solution. This analysis compares strategies learned in dynamic environments versus what would be optimal under static conditions.

6.1.1 Comparative Analysis Framework:

Dynamic Environment (Our Implementation)

- **Market Behavior**: Random transitions between Market States 1 and 2
- **Transition Probability**: 50% chance of each state daily
- **Strategic Requirement**: Policy must adapt to uncertainty
- **Optimal Approach**: Robust strategies that perform well across market conditions

Static Environment Scenarios (Theoretical Comparison) **Scenario A: Always Market State 1** - **Fixed Prices**: Product A = \$8, Product B = \$2 (permanent) - **Optimal Strategy**: Always maximize Product A production - **Decision Rule**: Simple heuristic - produce as much A as possible - **Risk**: Vulnerable if market conditions were to change

Scenario B: Always Market State 2
- **Fixed Prices**: Product A = \$3, Product B = \$5 (permanent) - **Optimal Strategy**: Always maximize Product B production - **Decision Rule**: Simple heuristic - focus entirely on Product B - **Risk**: Suboptimal if Product A becomes profitable

6.1.2 Key Research Questions:

1. **Value of Adaptation**: How much additional profit does the dynamic policy generate compared to static strategies?
2. **Robustness vs Specialization**: Does the dynamic policy sacrifice peak performance in specific conditions for overall robustness?
3. **Strategy Flexibility**: How does optimal behavior change when facing uncertainty versus certainty?
4. **Risk-Return Trade-offs**: Does uncertainty change the risk profile of optimal strategies?

6.1.3 Expected Insights:

- **Dynamic Premium**: Additional value from adapting to market changes
- **Hedging Strategies**: How optimal policy balances risk across market states

- **Information Value**: Worth of knowing market conditions will change
- **Strategic Robustness**: Stability of performance across different scenarios

[56]:
```python
def compare_policies():
    """
    DYNAMIC vs STATIC POLICY COMPARISON
    ===================================

    Compares the optimal policy learned in the dynamic environment
    (with changing market conditions) to what would be optimal if
    market conditions were always fixed.

    COMPARISON SCENARIOS:
    ====================
    1. DYNAMIC POLICY: Learned with market state transitions
    2. STATIC POLICY A: Assumes market is always HIGH_DEMAND_A
    3. STATIC POLICY B: Assumes market is always HIGH_DEMAND_B
    INSIGHTS:
    ========
    - How much value does adaptation provide?
    - What strategies work best under uncertainty?
    - How does optimal behavior change with and without market volatility?
    """
    print("\n" + "="*80)
    print("DYNAMIC vs STATIC POLICY COMPARISON")
    print("="*80)

    # Create environments for comparison
    print("\n1. SOLVING DYNAMIC ENVIRONMENT (changing market conditions)...")
    dynamic_env = SmartSupplierEnvironment()
    dynamic_solver = ValueIteration(dynamic_env)
    dynamic_values, dynamic_policy = dynamic_solver.solve()

    print("\n2. SOLVING STATIC ENVIRONMENT A (always HIGH_DEMAND_A)...")
    # For static environment, we'd need to modify the environment
    # This is a conceptual comparison - implementation would require
    # environment modification

    print("\n3. PERFORMANCE COMPARISON:")
    print("   (Note: Full implementation would require separate static␣
↪environments)")
    print("   Dynamic policy adapts to market changes")
    print("   Static policies would be suboptimal when market conditions␣
↪change")

    # Simulate dynamic policy
    simulator = PolicySimulator(dynamic_env, dynamic_policy)
    dynamic_results = simulator.run_simulation(1000)

    print(f"\nDYNAMIC POLICY PERFORMANCE:")
    print(f"   Average Profit: ${dynamic_results['mean_reward']:.2f}")
    print(f"   Adapts to changing market conditions")
    print(f"   Optimal for uncertain environment")
```

6.2 4.2 Dynamic vs Static Policy Comparison Implementation

The **compare_policies()** function implements a systematic comparison between our dynamic programming solution and hypothetical static strategies. This analysis quantifies the value of adaptation in uncertain environments.

6.2.1 Comparison Methodology:

Dynamic Policy (Our Solution)

- **Learning Environment**: Full Smart Supplier MDP with market uncertainty
- **Algorithm**: Value Iteration with stochastic transitions
- **Strategy**: Adapts decisions based on current market state
- **Performance**: Optimized for expected value across all market conditions

Static Policy Benchmarks

- **Static Policy A**: Assumes permanent Market State 1 (HIGH_DEMAND_A)
- **Static Policy B**: Assumes permanent Market State 2 (HIGH_DEMAND_B)
- **Limitation**: Cannot adapt when market conditions change
- **Risk**: Potentially catastrophic performance in wrong market state

6.2.2 Analysis Components:

1. Performance Metrics

- **Average Profit**: Expected return per 5-day episode
- **Profit Variance**: Risk and consistency of returns
- **Worst-Case Scenarios**: Performance in adverse conditions
- **Adaptation Speed**: How quickly policy responds to market changes

2. Strategic Insights

- **Hedging Behavior**: Does dynamic policy sacrifice peak performance for robustness?
- **Market Timing**: How effectively does policy exploit favorable conditions?
- **Resource Efficiency**: Optimal allocation under uncertainty vs certainty

3. Economic Value

- **Adaptation Premium**: Additional profit from dynamic strategy
- **Information Value**: Worth of knowing market conditions change
- **Risk Reduction**: Lower volatility through diversified strategies

6.2.3 Expected Outcomes:

- **Dynamic Superiority**: Better average performance in uncertain environment
- **Robustness Advantage**: More consistent performance across market conditions
- **Strategic Sophistication**: More nuanced decision-making patterns

[57]:
```python
# --- Main Execution ---
def main():
    """
    MAIN EXECUTION FUNCTION
    ======================
    Orchestrate the complete Dynamic Programming solution for the Smart␣
 ↪Supplier problem, including:
    1. Environment setup
    2. Value Iteration algorithm execution
    3. Policy analysis and interpretation
    4. Performance evaluation through simulation
    5. Comparative analysis

    This provides a comprehensive demonstration of Dynamic Programming
    applied to real-world optimization problems.
    """
    print("SMART SUPPLIER DYNAMIC PROGRAMMING SOLUTION")
    print("=" * 80)
    print("Optimizing production decisions under market uncertainty")
    print("Using Value Iteration algorithm for exact optimal policy")

    # STEP 1: ENVIRONMENT SETUP
    print("\n" + "="*50)
    print("STEP 1: ENVIRONMENT SETUP")
    print("="*50)
    env = SmartSupplierEnvironment()

    # STEP 2: SOLVE USING VALUE ITERATION
    print("\n" + "="*50)
    print("STEP 2: DYNAMIC PROGRAMMING SOLUTION")
    print("="*50)
    solver = ValueIteration(env)
    value_function, optimal_policy = solver.solve()

    # STEP 3: POLICY ANALYSIS
    print("\n" + "="*50)
    print("STEP 3: OPTIMAL POLICY ANALYSIS")
    print("="*50)
    analyzer = PolicyAnalyzer(env, value_function, optimal_policy)
    analyzer.analyze_market_dependency()
    analyzer.analyze_resource_dependency()
    analyzer.analyze_time_dependency()
    analyzer.analyze_value_function()

    # STEP 4: PERFORMANCE EVALUATION
    print("\n" + "="*50)
    print("STEP 4: PERFORMANCE EVALUATION")
    print("="*50)
    simulator = PolicySimulator(env, optimal_policy)
    simulation_results = simulator.run_simulation(1000)

    # STEP 5: COMPARATIVE ANALYSIS
    print("\n" + "="*50)
    print("STEP 5: COMPARATIVE ANALYSIS")
    print("="*50)
    compare_policies()

    # FINAL SUMMARY
    print("\n" + "="*80)
    print("DYNAMIC PROGRAMMING SOLUTION COMPLETE")
    print("="*80)
    print("Optimal policy learned using Value Iteration")
    print("Policy adapts to market conditions, resource constraints, and time␣
↪horizon")
    print("Performance validated through Monte Carlo simulation")
    print("Strategic insights extracted through comprehensive analysis")
    print(f"Expected profit: ${simulation_results['mean_reward']:.2f} per 5-day␣
↪period")

    return {
        'environment': env,
        'value_function': value_function,
        'optimal_policy': optimal_policy,
        'simulation_results': simulation_results
    }

if __name__ == "__main__":
    # Execute the complete Dynamic Programming solution
    results = main()
```

```
SMART SUPPLIER DYNAMIC PROGRAMMING SOLUTION
================================================================================
Optimizing production decisions under market uncertainty
Using Value Iteration algorithm for exact optimal policy

==================================================
STEP 1: ENVIRONMENT SETUP
==================================================
SMART SUPPLIER ENVIRONMENT INITIALIZED
==================================================
Episode length: 5 days
Daily raw materials: 10 units
Product A cost: 2 RM per unit
Product B cost: 1 RM per unit

Market Pricing:
  HIGH_DEMAND_A: A=$8, B=$2
  HIGH_DEMAND_B: A=$3, B=$5

==================================================
STEP 2: DYNAMIC PROGRAMMING SOLUTION
==================================================
VALUE ITERATION ALGORITHM INITIALIZED
========================================
State space size: 110
```

```
Discount factor: 1.0
Convergence tolerance: 1e-06
Maximum iterations: 1000

============================================================
STARTING VALUE ITERATION ALGORITHM
============================================================

Initializing value function…
Initialized 110 states with V(s) = 0

Iterating toward optimal value function…
  Iteration   1: Max change = 25.00000000
  Iteration   6: Max change = 0.00000000

  CONVERGED after 6 iterations!
  Final max change: 0.00e+00

Extracting optimal policy…
Extracted policy for 110 states

VALUE ITERATION COMPLETE
   Convergence achieved in 6 iterations

==================================================
STEP 3: OPTIMAL POLICY ANALYSIS
==================================================

======================================================================
MARKET STATE DEPENDENCY ANALYSIS
======================================================================

  OPTIMAL STRATEGY IN HIGH_DEMAND_A:
   Market Prices: A=$8, B=$2
   Action Distribution (55 states):
     PRODUCE_2A_0B   | 10 states ( 18.2%) | Produces: 2A, 0B
     PRODUCE_1A_2B   |  0 states (  0.0%) | Produces: 1A, 2B
     PRODUCE_0A_5B   |  0 states (  0.0%) | Produces: 0A, 5B
     PRODUCE_3A_0B   | 25 states ( 45.5%) | Produces: 3A, 0B
     DO_NOTHING      | 20 states ( 36.4%) | Produces: 0A, 0B

  OPTIMAL STRATEGY IN HIGH_DEMAND_B:
   Market Prices: A=$3, B=$5
   Action Distribution (55 states):
     PRODUCE_2A_0B   |  0 states (  0.0%) | Produces: 2A, 0B
     PRODUCE_1A_2B   |  5 states (  9.1%) | Produces: 1A, 2B
     PRODUCE_0A_5B   | 30 states ( 54.5%) | Produces: 0A, 5B
     PRODUCE_3A_0B   |  0 states (  0.0%) | Produces: 3A, 0B
```

```
    DO_NOTHING      | 20 states ( 36.4%) | Produces: 0A, 0B

================================================================================
RESOURCE CONSTRAINT IMPACT ANALYSIS
================================================================================

  STRATEGY WITH 0 RAW MATERIALS:
   Action Distribution (10 states):
     DO_NOTHING      | 10 states (100.0%) | Cost: 0RM

  STRATEGY WITH 2 RAW MATERIALS:
   Action Distribution (10 states):
     DO_NOTHING      | 10 states (100.0%) | Cost: 0RM

  STRATEGY WITH 4 RAW MATERIALS:
   Action Distribution (10 states):
     PRODUCE_2A_0B   |  5 states ( 50.0%) | Cost: 4RM
     PRODUCE_1A_2B   |  5 states ( 50.0%) | Cost: 4RM

  STRATEGY WITH 6 RAW MATERIALS:
   Action Distribution (10 states):
     PRODUCE_0A_5B   |  5 states ( 50.0%) | Cost: 5RM
     PRODUCE_3A_0B   |  5 states ( 50.0%) | Cost: 6RM

  STRATEGY WITH 8 RAW MATERIALS:
   Action Distribution (10 states):
     PRODUCE_0A_5B   |  5 states ( 50.0%) | Cost: 5RM
     PRODUCE_3A_0B   |  5 states ( 50.0%) | Cost: 6RM

  STRATEGY WITH 10 RAW MATERIALS:
   Action Distribution (10 states):
     PRODUCE_0A_5B   |  5 states ( 50.0%) | Cost: 5RM
     PRODUCE_3A_0B   |  5 states ( 50.0%) | Cost: 6RM

================================================================================
TIME HORIZON EFFECTS ANALYSIS
================================================================================

  DAY 1 STRATEGY:
   Average State Value: $112.32
   Action Distribution (22 states):
     PRODUCE_2A_0B   |  2 states (  9.1%) | Produces: 2A, 0B
     PRODUCE_1A_2B   |  1 states (  4.5%) | Produces: 1A, 2B
     PRODUCE_0A_5B   |  6 states ( 27.3%) | Produces: 0A, 5B
     PRODUCE_3A_0B   |  5 states ( 22.7%) | Produces: 3A, 0B
     DO_NOTHING      |  8 states ( 36.4%) | Produces: 0A, 0B

  DAY 2 STRATEGY:
   Average State Value: $87.82
   Action Distribution (22 states):
     PRODUCE_2A_0B   |  2 states (  9.1%) | Produces: 2A, 0B
     PRODUCE_1A_2B   |  1 states (  4.5%) | Produces: 1A, 2B
     PRODUCE_0A_5B   |  6 states ( 27.3%) | Produces: 0A, 5B
     PRODUCE_3A_0B   |  5 states ( 22.7%) | Produces: 3A, 0B
     DO_NOTHING      |  8 states ( 36.4%) | Produces: 0A, 0B

  DAY 3 STRATEGY:
   Average State Value: $63.32
   Action Distribution (22 states):
     PRODUCE_2A_0B   |  2 states (  9.1%) | Produces: 2A, 0B
     PRODUCE_1A_2B   |  1 states (  4.5%) | Produces: 1A, 2B
     PRODUCE_0A_5B   |  6 states ( 27.3%) | Produces: 0A, 5B
     PRODUCE_3A_0B   |  5 states ( 22.7%) | Produces: 3A, 0B
     DO_NOTHING      |  8 states ( 36.4%) | Produces: 0A, 0B

  DAY 4 STRATEGY:
   Average State Value: $38.82
   Action Distribution (22 states):
     PRODUCE_2A_0B   |  2 states (  9.1%) | Produces: 2A, 0B
     PRODUCE_1A_2B   |  1 states (  4.5%) | Produces: 1A, 2B
     PRODUCE_0A_5B   |  6 states ( 27.3%) | Produces: 0A, 5B
     PRODUCE_3A_0B   |  5 states ( 22.7%) | Produces: 3A, 0B
     DO_NOTHING      |  8 states ( 36.4%) | Produces: 0A, 0B

  DAY 5 STRATEGY:
   Average State Value: $14.32
   Action Distribution (22 states):
     PRODUCE_2A_0B   |  2 states (  9.1%) | Produces: 2A, 0B
     PRODUCE_1A_2B   |  1 states (  4.5%) | Produces: 1A, 2B
     PRODUCE_0A_5B   |  6 states ( 27.3%) | Produces: 0A, 5B
     PRODUCE_3A_0B   |  5 states ( 22.7%) | Produces: 3A, 0B
     DO_NOTHING      |  8 states ( 36.4%) | Produces: 0A, 0B

======================================================================
VALUE FUNCTION ANALYSIS
======================================================================

TOP 10 MOST VALUABLE STATES:
    1. Day 1, RM= 5, HIGH_DEMAND_B   | V=$123.00 | Action: 0A,5B
    2. Day 1, RM= 6, HIGH_DEMAND_B   | V=$123.00 | Action: 0A,5B
    3. Day 1, RM= 7, HIGH_DEMAND_B   | V=$123.00 | Action: 0A,5B
    4. Day 1, RM= 8, HIGH_DEMAND_B   | V=$123.00 | Action: 0A,5B
    5. Day 1, RM= 9, HIGH_DEMAND_B   | V=$123.00 | Action: 0A,5B
    6. Day 1, RM=10, HIGH_DEMAND_B   | V=$123.00 | Action: 0A,5B
    7. Day 1, RM= 6, HIGH_DEMAND_A   | V=$122.00 | Action: 3A,0B
    8. Day 1, RM= 7, HIGH_DEMAND_A   | V=$122.00 | Action: 3A,0B
    9. Day 1, RM= 8, HIGH_DEMAND_A   | V=$122.00 | Action: 3A,0B
   10. Day 1, RM= 9, HIGH_DEMAND_A   | V=$122.00 | Action: 3A,0B

BOTTOM 10 LEAST VALUABLE STATES:
    1. Day 5, RM= 5, HIGH_DEMAND_A   | V=$ 16.00 | Action: 2A,0B
    2. Day 5, RM= 4, HIGH_DEMAND_B   | V=$ 13.00 | Action: 1A,2B
    3. Day 5, RM= 0, HIGH_DEMAND_A   | V=$  0.00 | Action: 0A,0B
    4. Day 5, RM= 0, HIGH_DEMAND_B   | V=$  0.00 | Action: 0A,0B
    5. Day 5, RM= 1, HIGH_DEMAND_A   | V=$  0.00 | Action: 0A,0B
    6. Day 5, RM= 1, HIGH_DEMAND_B   | V=$  0.00 | Action: 0A,0B
    7. Day 5, RM= 2, HIGH_DEMAND_A   | V=$  0.00 | Action: 0A,0B
    8. Day 5, RM= 2, HIGH_DEMAND_B   | V=$  0.00 | Action: 0A,0B
    9. Day 5, RM= 3, HIGH_DEMAND_A   | V=$  0.00 | Action: 0A,0B
   10. Day 5, RM= 3, HIGH_DEMAND_B   | V=$  0.00 | Action: 0A,0B

VALUE FUNCTION STATISTICS:
   Maximum Value: $123.00
   Minimum Value: $0.00
   Average Value: $63.32
   Standard Deviation: $36.44

==================================================
STEP 4: PERFORMANCE EVALUATION
==================================================

======================================================================
POLICY SIMULATION: 1000 EPISODES
======================================================================
  Episode  100: Running average = $122.39
  Episode  200: Running average = $122.57
  Episode  300: Running average = $122.57
  Episode  400: Running average = $122.58
  Episode  500: Running average = $122.59
  Episode  600: Running average = $122.54
  Episode  700: Running average = $122.55
  Episode  800: Running average = $122.53
  Episode  900: Running average = $122.54
  Episode 1000: Running average = $122.53

SIMULATION RESULTS SUMMARY:
   Episodes Simulated: 1000
   Average Profit: $122.53 ± $1.11
   95% Confidence Interval: $122.46 - $122.60
   Profit Range: $120.00 - $125.00

ACTION USAGE DISTRIBUTION:
   PRODUCE_2A_0B   |   0.0% | Produces: 2A, 0B
   PRODUCE_1A_2B   |   0.0% | Produces: 1A, 2B
   PRODUCE_0A_5B   |  50.6% | Produces: 0A, 5B
   PRODUCE_3A_0B   |  49.4% | Produces: 3A, 0B
   DO_NOTHING      |   0.0% | Produces: 0A, 0B

==================================================
STEP 5: COMPARATIVE ANALYSIS
==================================================

================================================================================
DYNAMIC vs STATIC POLICY COMPARISON
================================================================================

1. SOLVING DYNAMIC ENVIRONMENT (changing market conditions)…
SMART SUPPLIER ENVIRONMENT INITIALIZED
==================================================
Episode length: 5 days
Daily raw materials: 10 units
Product A cost: 2 RM per unit
Product B cost: 1 RM per unit

Market Pricing:
  HIGH_DEMAND_A: A=$8, B=$2
  HIGH_DEMAND_B: A=$3, B=$5
VALUE ITERATION ALGORITHM INITIALIZED
========================================
State space size: 110
Discount factor: 1.0
Convergence tolerance: 1e-06
Maximum iterations: 1000

============================================================
STARTING VALUE ITERATION ALGORITHM
============================================================

Initializing value function…
Initialized 110 states with V(s) = 0

Iterating toward optimal value function…
  Iteration   1: Max change = 25.00000000
  Iteration   6: Max change = 0.00000000

 CONVERGED after 6 iterations!
  Final max change: 0.00e+00

Extracting optimal policy…
Extracted policy for 110 states

VALUE ITERATION COMPLETE
```

```
   Convergence achieved in 6 iterations

2. SOLVING STATIC ENVIRONMENT A (always HIGH_DEMAND_A)…

3. PERFORMANCE COMPARISON:
   (Note: Full implementation would require separate static environments)
   Dynamic policy adapts to market changes
   Static policies would be suboptimal when market conditions change

======================================================================
POLICY SIMULATION: 1000 EPISODES
======================================================================
  Episode  100: Running average = $122.58
  Episode  200: Running average = $122.52
  Episode  300: Running average = $122.54
  Episode  400: Running average = $122.56
  Episode  500: Running average = $122.53
  Episode  600: Running average = $122.51
  Episode  700: Running average = $122.51
  Episode  800: Running average = $122.48
  Episode  900: Running average = $122.48
  Episode 1000: Running average = $122.49

SIMULATION RESULTS SUMMARY:
   Episodes Simulated: 1000
   Average Profit: $122.49 ± $1.13
   95% Confidence Interval: $122.42 - $122.56
   Profit Range: $120.00 - $125.00

ACTION USAGE DISTRIBUTION:
   PRODUCE_2A_0B   |   0.0% | Produces: 2A, 0B
   PRODUCE_1A_2B   |   0.0% | Produces: 1A, 2B
   PRODUCE_0A_5B   |  49.7% | Produces: 0A, 5B
   PRODUCE_3A_0B   |  50.3% | Produces: 3A, 0B
   DO_NOTHING      |   0.0% | Produces: 0A, 0B

DYNAMIC POLICY PERFORMANCE:
   Average Profit: $122.49
   Adapts to changing market conditions
   Optimal for uncertain environment

================================================================================
DYNAMIC PROGRAMMING SOLUTION COMPLETE
================================================================================
Optimal policy learned using Value Iteration
Policy adapts to market conditions, resource constraints, and time horizon
Performance validated through Monte Carlo simulation
Strategic insights extracted through comprehensive analysis
```

```
Expected profit: $122.53 per 5-day period
```

6.3 6. Conclusion and Key Learnings

6.3.1 Key Technical Accomplishments:

1. Complete MDP Formulation

- **State Space**: 110 states capturing day, resources, and market conditions
- **Action Space**: 5 strategic production decisions with resource constraints

- **Transition Dynamics**: Stochastic market changes with deterministic day progression
- **Reward Structure**: Profit-based rewards reflecting real economic incentives

2. Exact Optimal Solution

- **Value Iteration Algorithm**: Mathematically guaranteed optimal policy
- **Convergence Verification**: Algorithm reaches stable solution within tolerance
- **Policy Extraction**: Complete optimal strategy for all possible states
- **Performance Validation**: Theoretical predictions confirmed through simulation

3. Comprehensive Analysis

- **Market Adaptation**: Policy successfully adapts to changing conditions
- **Resource Management**: Efficient allocation under scarcity constraints
- **Time Sensitivity**: Strategic evolution as deadlines approach
- **Economic Rationality**: Decisions align with profit maximization principles

6.3.2 Strategic Insights Discovered:

Market Intelligence

- Optimal policy demonstrates clear market awareness
- Favors Product A during HIGH_DEMAND_A states ($8 vs $2 pricing)
- Switches to Product B focus during HIGH_DEMAND_B states ($5 vs $3 pricing)
- **Lesson**: Adaptation to changing conditions creates significant value

Resource Optimization

- Low raw materials drive conservative, efficient strategies
- High raw materials enable aggressive profit maximization
- Policy balances immediate production with constraint respect
- **Lesson**: Scarcity demands intelligent resource allocation

Time Horizon Effects

- Early days favor balanced, risk-conscious approaches
- Final days justify more aggressive strategies
- Finite horizon eliminates infinite future considerations
- **Lesson**: Deadline proximity affects optimal risk tolerance

6.3.3 Value of Dynamic Programming:

Theoretical Guarantees

- **Global Optimality**: Proven to find best possible strategy
- **Mathematical Rigor**: Based on solid theoretical foundations
- **Convergence Properties**: Guaranteed to reach optimal solution
- **No Local Optima**: Avoids getting stuck in suboptimal solutions

Practical Advantages

- **Complete Solution**: Provides strategy for all possible situations
- **Uncertainty Handling**: Explicitly accounts for stochastic elements
- **Constraint Integration**: Naturally incorporates resource limitations
- **Performance Transparency**: Clear understanding of why decisions are optimal

6.3.4 Real-World Applications:

This Smart Supplier example represents a class of problems found throughout business and industry: - **Supply Chain Management**: Production planning under demand uncertainty - **Financial Planning**: Portfolio allocation with market volatility - **Resource Allocation**: Capacity planning with stochastic demand - **Strategic Planning**: Multi-period decision making under uncertainty

6.3.5 Educational Value:

Students completing this implementation gain deep understanding of: - **MDP Formulation**: How to model real problems as mathematical frameworks - **Algorithm Implementation**: Translating theory into working code - **Performance Analysis**: Validating solutions through comprehensive testing - **Strategic Thinking**: Understanding how optimal strategies emerge from mathematical optimization —