Group ID: 302

Group Members Name with Student ID:

1. KARTHIKEYAN J - 2024AA05372
2. JANGALE SAVEDANA SUBHASH PRATIBHA - 2024AA05187
3. GANAPATHY SUBRAMANIAN S - 2024AA05188
4. ANANDAN A - 2024AA05269

The Smart Supplier: Optimizing Orders in a Fluctuating Market - 6 Marks

Develop a reinforcement learning agent using dynamic programming to help a Smart Supplier decide which products to manufacture and sell each day to maximize profit. The agent must learn the optimal policy for choosing daily production quantities, considering its limited raw materials and the unpredictable daily demand and selling prices for different products.

Scenario

A small Smart Supplier manufactures two simple products: Product A and Product B. Each day, the supplier has a limited amount of raw material. The challenge is that the market demand and selling price for Product A and Product B change randomly each day, making some products more profitable than others at different times. The supplier needs to decide how much of each product to produce to maximize profit while managing their limited raw material.

Objective

The Smart Supplier's agent must learn the optimal policy $\pi*$ using dynamic programming (Value Iteration or Policy Iteration) to decide how many units of Product A and Product B to produce each day to maximize the total profit over the fixed number of days, given the daily changing market conditions and limited raw material.

--- 1. Custom Environment Creation (SmartSupplierEnv) --- (1 Mark)

```
# Define market states and their product prices
# Structure: {Market_State_ID: {'A_price': X, 'B_price': Y}}
# Define product raw material costs
# 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

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

```
# get reward function

import numpy as np
import random

class SmartSupplierEnv:
    """
    Custom environment for the Smart Supplier problem.
    Handles state transitions, actions, market states, and reward 
calculation.
    """

    def __init__(self, num_days=5, initial_rm=10):
        self.num_days = num_days
        self.initial_rm = initial_rm

        # Define action space with associated costs
        self.actions = {
            0: (2, 0, 4),  # Produce 2A, 0B → Costs 4 RM
            1: (1, 2, 4),  # Produce 1A, 2B → Costs 4 RM
            2: (0, 5, 5),  # Produce 0A, 5B → Costs 5 RM
            3: (3, 0, 6),  # Produce 3A, 0B → Costs 6 RM
            4: (0, 0, 0),  # Do Nothing → Costs 0 RM
        }

        # Define market states with associated prices
        self.market_states = {
            1: {'A_price': 8, 'B_price': 2},  # High demand for A
            2: {'A_price': 3, 'B_price': 5},  # High demand for B
        }

    def get_all_states(self):
        """
        Generate all valid states in the environment.
        Each state is a tuple: (current_day, remaining_raw_material, 
market_state)
        """
        return [(day, rm, mkt)
                for day in range(1, self.num_days + 1)
                for rm in range(0, self.initial_rm + 1)
                for mkt in [1, 2]]

    def get_possible_actions(self, rm_left):
        """
        Return list of action IDs possible given current raw material.
        """
        return [a for a, (_, _, cost) in self.actions.items() if cost 
<= rm_left]
```

```
    def get_reward(self, action_id, market_state):
        """
        Calculate profit earned from a given action in a specific
market state.
        """
        a_units, b_units, _ = self.actions[action_id]
        prices = self.market_states[market_state]
        return a_units * prices['A_price'] + b_units *
prices['B_price']

    def transition(self, state, action_id):
        """
        Perform transition from one state to next state based on
action.
        Returns (next_state, reward). If terminal, next_state is None.
        """
        day, rm, market = state
        _, _, cost = self.actions[action_id]

        if cost > rm:
            return state, 0  # Invalid move: return same state, zero
reward

        reward = self.get_reward(action_id, market)

        # Move to next day, reset raw material
        next_day = day + 1
        if next_day > self.num_days:
            return None, reward  # End of episode

        next_market = random.choice([1, 2])
        return (next_day, self.initial_rm, next_market), reward
```

--- 2. Dynamic Programming Implementation (Value Iteration or Policy Iteration) --- (2 Mark)

```
# Value Iteration function

def value_iteration(env, gamma=1.0, theta=1e-4):
    """
    Performs Value Iteration to compute the optimal policy.
    Returns:
        V: State-value function (dict)
        policy: Optimal action for each state (dict)
    """
    V = {state: 0 for state in env.get_all_states()}
    policy = {}

    iteration = 0
    while True:
```

```
        delta = 0
        new_V = V.copy()

        for state in env.get_all_states():
            day, rm, mkt = state
            if day > env.num_days:
                continue  # Terminal state

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

            for action in env.get_possible_actions(rm):
                total_value = 0
                reward = env.get_reward(action, mkt)

                # For value iteration, consider future market states
                for next_market in [1, 2]:
                    next_day = day + 1
                    if next_day > env.num_days:
                        future_val = 0
                    else:
                        next_state = (next_day, env.initial_rm,
next_market)
                        future_val = V[next_state]
                    prob = 0.5  # Market transitions are equally
likely
                    total_value += prob * (reward + gamma *
future_val)

                if total_value > best_action_value:
                    best_action_value = total_value
                    best_action = action

            new_V[state] = best_action_value
            policy[state] = best_action
            delta = max(delta, abs(V[state] - best_action_value))

        V = new_V
        iteration += 1

        if delta < theta:
            break

    print(f"Value iteration converged in {iteration} iterations.")
    return V, policy
```

--- 3. Simulation and Policy Analysis --- (1 Mark)

```
# simulate policy function - Simulates the learned policy over
multiple runs to evaluate performance
```

```python
def simulate_policy(env, policy, runs=1000):
    """
    Simulate the policy over multiple runs to estimate average total reward.
    """
    total_rewards = []

    for _ in range(runs):
        state = (1, env.initial_rm, random.choice([1, 2]))
        total_reward = 0

        while state is not None:
            action = policy.get(state, 4)  # Default to 'Do Nothing' if state not found
            next_state, reward = env.transition(state, action)
            total_reward += reward
            state = next_state

        total_rewards.append(total_reward)

    average_reward = np.mean(total_rewards)
    print(f"Average total reward over {runs} runs: ${average_reward:.2f}")
    return average_reward

# analyze policy function - Analyzes and prints snippets of the learned optimal policy

def analyze_policy(policy, env):
    """
    Analyzes how the learned policy behaves across different conditions.
    """
    print("Policy Analysis for Day 1 with 10 RM:")
    for market in [1, 2]:
        state = (1, 10, market)
        action = policy.get(state)
        a_units, b_units, _ = env.actions[action]
        print(f"  Market State {market}: Action → {a_units}A and {b_units}B")

    print("\nPolicy behavior with low RM:")
    for rm in range(0, 6):
        state = (1, rm, 1)
        action = policy.get(state, None)
        if action is not None:
            a_units, b_units, _ = env.actions[action]
            print(f"  RM = {rm}: Action → {a_units}A and {b_units}B")
```

```
    print("\nPolicy on last day (Day 5):")
    for market in [1, 2]:
        state = (5, 10, market)
        action = policy.get(state)
        a_units, b_units, _ = env.actions[action]
        print(f"  Market {market}: Action → {a_units}A and 
{b_units}B")
```

--- 4. Impact of Dynamics Analysis --- (1 Mark)

```
# Discusses the impact of dynamic market prices on the optimal policy.
# This section should primarily be a written explanation in the 
report.

# In the markdown below.
```

Dynamic Market Impact Discussion (To include in your markdown/report):

If the market state were fixed (e.g., always Market State 1), the optimal policy would favor producing Product A heavily due to its consistently higher reward.

However, because the market changes randomly each day, the policy learned using dynamic programming becomes adaptive. It accounts for both possibilities and spreads risk—choosing more flexible or balanced production when uncertainty is high.

On the last day, the policy tends to be more aggressive, using up all available resources since there's no future to plan for. This is a clear indicator of the dynamic programming approach considering the finite time horizon in decision-making.

```
# --- Main Execution ---

env = SmartSupplierEnv()
V, policy = value_iteration(env)
simulate_policy(env, policy)
analyze_policy(policy, env)
```

```
Value iteration converged in 6 iterations.
Average total reward over 1000 runs: $122.49
Policy Analysis for Day 1 with 10 RM:
  Market State 1: Action → 3A and 0B
  Market State 2: Action → 0A and 5B

Policy behavior with low RM:
  RM = 0: Action → 0A and 0B
  RM = 1: Action → 0A and 0B
```

```
  RM = 2: Action → 0A and 0B
  RM = 3: Action → 0A and 0B
  RM = 4: Action → 2A and 0B
  RM = 5: Action → 2A and 0B

Policy on last day (Day 5):
  Market 1: Action → 3A and 0B
  Market 2: Action → 0A and 5B
```