DAA PROJECT

Problem Statement

Title: Optimal Investment Strategy Using Markov Decision Process

Given an economic environment with three distinct market conditions (bull market, bear market, and stagnant market), we aim to determine the optimal investment strategy between stocks and bonds. Each investment choice comes with certain rewards and transition probabilities to different market conditions. Our objective is to maximize the expected cumulative reward over time by identifying the optimal policy using the value iteration algorithm, a dynamic programming technique.

Abstract

In a dynamic economic environment, investors need to make informed decisions about their investment strategies to maximize returns. This study presents a method for determining the optimal investment strategy using a Markov Decision Process (MDP). We consider three market states: bull market, bear market, and stagnant market, and two possible actions: investing in stocks or bonds. By modeling the problem as an MDP, we use the value iteration algorithm to compute the optimal value function and extract the corresponding optimal policy. The results indicate the

best investment action for each market state to maximize long-term rewards, demonstrating the practical application of dynamic programming in financial decision-making.

Design Strategy

Algorithm Used: Dynamic Programming

1. Define the MDP Components

- **States**: Represent the different market conditions. We have three states: bull market, bear market, and stagnant market.
- **Actions**: Represent the investment choices. We have two actions: invest in stocks and invest in bonds.
- **Transition Probabilities**: Represent the probability of transitioning from one state to another given a specific action.
- **Rewards**: Represent the immediate returns from taking a specific action in a given state.
- **Discount Factor**: A factor to discount future rewards, ensuring current rewards are valued more highly.

2. Initialize the Transition and Reward Matrices

- **Transition Matrix**: A 3D matrix where `transitionMatrix[action][state][nextState]` indicates the probability of moving from `state` to `nextState` when taking `action`.

- **Reward Matrix**: A 2D matrix where `rewardMatrix[action][state]` indicates the reward for taking `action` in `state`.

3. Implement Value Iteration Algorithm

- **Initialize Value Function**: Start with an initial guess for the value of each state (e.g., all zeros).
- **Iteratively Update Values**: For each state, calculate the expected value of each action, considering immediate rewards and future values, and update the value function accordingly.
- **Check for Convergence**: Continue the iterative process until the change in value function (`delta`) is below a defined threshold.

4. Extract the Optimal Policy

- **Determine Best Action**: For each state, compute the action that maximizes the expected value based on the converged value function.
- **Store Policy**: Save the best action for each state in a policy vector.

5. Output the Results

- **Print Optimal Value Function:** Display the computed value for each state.
- **Print Optimal Policy**: Indicate the best action (stocks or bonds) for each market state.

6. Main Function

- **Integrate Components**: Call the value iteration function to compute the optimal value function, extract the optimal policy, and print the results.

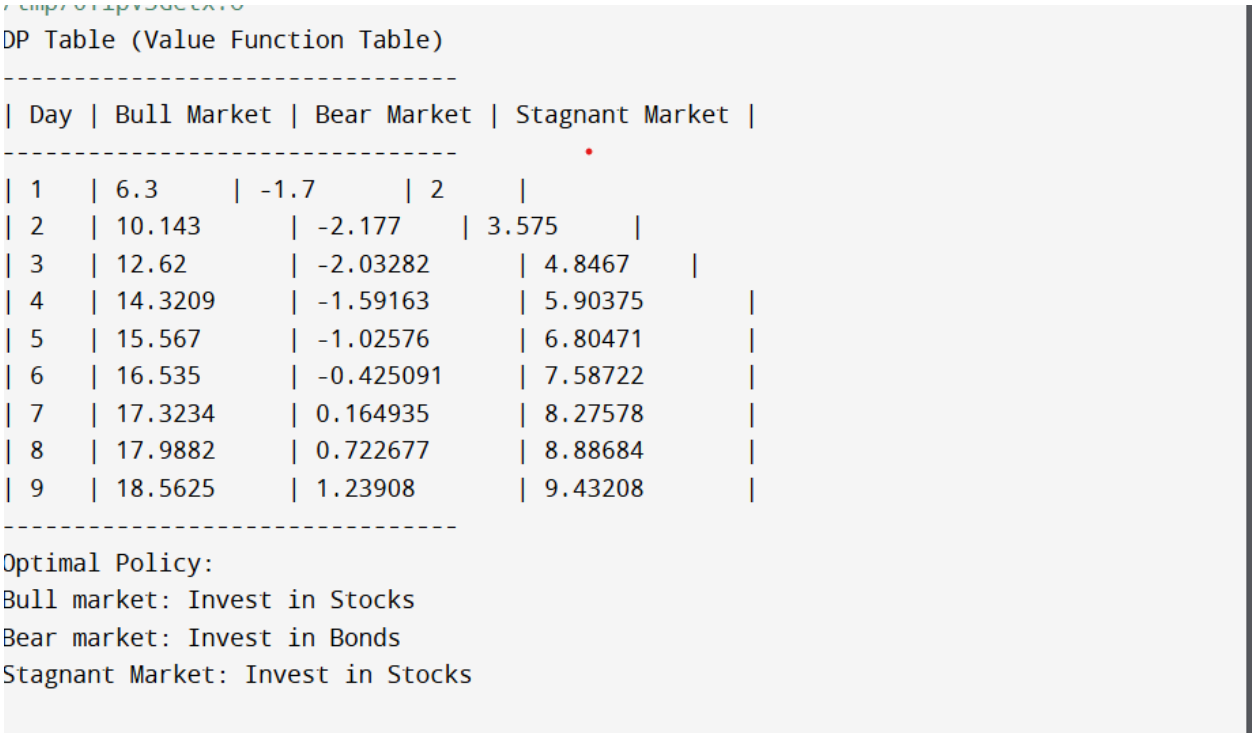
Detailed Code with Explanations

```
#include <iostream>

#include <vector>
#include <cmath>
#include <algorithm>

using namespace std;

const int numStates = 3;
const int numActions = 2;
const double discountFactor = 0.9;
const double threshold = 1e-6;

vector<vector<vector<double>>> transitionMatrix = {
    { {0.7, 0.2, 0.1}, {0.1, 0.8, 0.1}, {0.2, 0.3,
0.5} },
    { {0.1, 0.8, 0.1}, {0.1, 0.8, 0.1}, {0.2, 0.3,
0.5} }
};

vector<vector<double>> rewardMatrix = {
    {10, -5, 3},
    {5, -3, 2}
```

```
};

vector<double> valueIteration() {
    vector<double> valueFunction(numStates, 0.0);
    vector<double> newValueFunction(numStates, 0.0);

    cout << "DP Table (Value Function Table)" <<
endl;
    cout << "--------------------------------" <<
endl;
    cout << "| Day | Bull Market | Bear Market |
Stagnant Market |" << endl;
    cout << "--------------------------------" <<
endl;
    int day ;
    for(day=1;day<10;day++) {
        double delta = 0.0;

        for (int state = 0; state < numStates;
++state) {
            double maxActionValue = -INFINITY;

            for (int action = 0; action < numActions;
++action) {
                double actionValue = 0.0;

                for (int nextState = 0; nextState <
numStates; ++nextState) {
                    actionValue +=
transitionMatrix[action][state][nextState] *
(rewardMatrix[action][nextState] + discountFactor *
valueFunction[nextState]);
                }

                if (actionValue > maxActionValue) {
                    maxActionValue = actionValue;
                }
            }

            newValueFunction[state] = maxActionValue;
            delta = max(delta,
fabs(newValueFunction[state] -
valueFunction[state]));
        }

        valueFunction = newValueFunction;

        cout << "| " << day << "   ";
        for (int state = 0; state < numStates;
++state) {
            cout << "| " << valueFunction[state] << "
\t";
        }
        cout << "|" << endl;

    }

    cout << "--------------------------------" <<
endl;

    return valueFunction;
```

```
}

vector<int> extractPolicy(const vector<double>&
valueFunction) {
    vector<int> policy(numStates, 0);

    for (int state = 0; state < numStates; ++state) {
        double maxActionValue = -INFINITY;
        int bestAction = 0;

        for (int action = 0; action < numActions;
++action) {
            double actionValue = 0.0;

            for (int nextState = 0; nextState <
numStates; ++nextState) {
                actionValue +=
transitionMatrix[action][state][nextState] *
(rewardMatrix[action][nextState] + discountFactor *
valueFunction[nextState]);
            }

            if (actionValue > maxActionValue) {
                maxActionValue = actionValue;
                bestAction = action;
            }
        }

        policy[state] = bestAction;
    }
    return policy;
}

int main() {
    vector<double> optimalValueFunction =
valueIteration();
    vector<int> optimalPolicy =
extractPolicy(optimalValueFunction);
    string s[3]={"Bull market", "Bear
market","Stagnant Market"};

    cout << "Optimal Policy:" << endl;
    for (int state = 0; state < numStates; ++state) {
        cout  << s[state] << ": " <<
(optimalPolicy[state] == 0 ? "Invest in Stocks" :
"Invest in Bonds") << endl;
    }

    return 0;
}
```

OUTPUT :



```
DP Table (Value Function Table)
--------------------------------
| Day | Bull Market | Bear Market | Stagnant Market |
--------------------------------
| 1   | 6.3     | -1.7      | 2     |
| 2   | 10.143      | -2.177    | 3.575     |
| 3   | 12.62       | -2.03282      | 4.8467    |
| 4   | 14.3209     | -1.59163      | 5.90375       |
| 5   | 15.567      | -1.02576      | 6.80471       |
| 6   | 16.535      | -0.425091     | 7.58722       |
| 7   | 17.3234     | 0.164935      | 8.27578       |
| 8   | 17.9882     | 0.722677      | 8.88684       |
| 9   | 18.5625     | 1.23908       | 9.43208       |
--------------------------------
Optimal Policy:
Bull market: Invest in Stocks
Bear market: Invest in Bonds
Stagnant Market: Invest in Stocks
```

Time Complexity: O(n^3)

Space Complexity: O(n^3)

Conclusion

This project demonstrates the use of dynamic programming through the value iteration algorithm to solve a Markov Decision Process for optimal investment strategy. By iteratively refining the value function and extracting the optimal policy, we can make informed decisions that maximize long-term rewards in different market conditions. This approach showcases the power of dynamic programming in solving complex decision-making problems. Thank you!