Group - 21

Gautam Kumar Mahar (2103114)

Kanwar raj Singh (1903122)

CS330 Artificial Intelligence- Lab 5

You are expected to code from scratch in Python or C or C++.

Consider the standard gambler's problem where a gambler can make bets on the outcomes of a sequence of coin flips. If the coin comes up heads, he wins as many units as he has staked. If it is tails, he loses what he has staked. The coin comes up with heads with a probability p (which should be tunable). The game ends when the gambler has reached a N units of capital (N should again be tunable) or loses by having zero capital. On each flip the gambler has to decide the portion of his capital to stake (in integers). If he makes N units of capital, he will obtain a one-time reward of 2N units and the game ends. The problem can be modeled as an MDP, where state is the gambler's capital $s \in \{0, 1, 2, . . . , N\}$, 0 and N are terminal states. Actions are stakes $a \in \{0, 1, . . . , \min(s, N - s)\}$. Every transition has a zero reward except for when the gambler reaches state N where a one time reward of 2N is obtained.

Experiment Setup:

N (Capital): 10

Probability of getting heads (p): 0.4

Discount factor (γ): 0.9

Policies:

a) Minimum Step Policy (Policy 1):

Policy: $\pi(s) = \min(s, N - s)$ for all states s.

Resulting values for states under this policy are calculated.

b) Fixed Step Policy (Policy 2):

Policy: $\pi(s) = 1$ for all states s.

Resulting values for states under this policy are calculated.

Value Iteration:

We implement the value iteration algorithm to find the optimal policy.

The optimal policy and value function are reported.

Policy Iteration:

We implement the policy iteration algorithm to find the optimal policy.

The optimal policy and value of the policy are reported.

Results:

Policy 1 (Minimum Step Policy):

This policy selects the minimum step to reach either end (0 or N).

The values for states under this policy are computed.

Policy 2 (Fixed Step Policy):

This policy always selects a fixed step size of 1.

The values for states under this policy are computed.

```
┌──(gautamop㉿gautamop)-[~/Desktop/CS330/LAB_ASSIGNMENT_3A]
└─$ cd "/home/gautamop/Desktop/CS330/LAB_ASSIGNMENT_3A/" && g++ LAB_ASSIGN_3A_GROUP_21.cpp -o LAB_ASSIGN_3A_GROUP_21 && "/home/gautamop/Deskt
CS330/LAB_ASSIGNMENT_3A/"LAB_ASSIGN_3A_GROUP_21
Using Minimum Step Policy:
State | Action | Utility
0 -1 0
1 1 0.53764
2 2 1.49344
3 3 2.88233
4 4 4.14846
5 5 7.2
6 4 8.00646
7 3 9.44017
8 2 11.5235
9 1 13.4227
10 -1 20

Using Fixed Step Policy:
State | Action | Utility
0 -1 0
1 1 0.0205948
2 1 0.0572078
3 1 0.128018
4 1 0.269795
5 1 0.557403
6 1 1.14365
7 1 2.3407
8 1 4.78647
9 1 9.78469
10 -1 20
```

2. Implement value iteration algorithm to find the optimal policy. Report the optimal policy and optimal value function you got.

```
LAB_ASSIGN_3A_GROUP_21.cpp > ...
184   }
PROBLEMS   OUTPUT   TERMINAL   DEBUG CONSOLE
7 1 2.3407
8 1 4.78647
9 1 9.78469
10 -1 20

Optimal Policy using Value Iteration:
0 -1 0
1 1 0.53764
2 2 1.49344
3 2 2.88233
4 1 4.14846
5 5 7.2
6 4 8.00646
7 3 9.44017
8 2 11.5235
9 1 13.4227
10 -1 20
```

3. Implement policy iteration algorithm to find the optimal policy. Report the optimal policy and value of the policy you got.

```
Optimal Policy Using Policy Iteration:
0 -1 0
1 1 0.53764
2 2 1.49344
3 2 2.88233
4 1 4.14846
5 5 7.2
6 4 8.00646
7 3 9.44017
8 2 11.5235
9 1 13.4227
10 -1 20
```

You may take N = 10, p = 0.4 and γ = 0.9 to answer the questions, however please experiment with different values of p, larger values of N (say upto N = 100 at least) and 0 < γ < 1. Also experiment with how the optimal policy might change if every transition (other than transition to state N) incurs a reward of −1.

```cpp
/*
                                                    [ Group Number : 21 ]
                                                      GAUTAM KUMAR MAHAR
                                                       KanwarRaj Singh

*/

#include <iostream>
#include <cstdlib>
#include <vector>
#include <cmath>
using namespace std;


double headProbability = 0.4; // Probability of getting heads

int numStates = 10; // Number of states

int maxReward = numStates * 2; // Maximum possible reward

double discountFactor = 0.9; // Discount factor

int stateCount = numStates + 1; // Number of states including terminal states

double epsilon = 1e-14; // Convergence threshold
```

```cpp
// Define a policy that selects the minimum of steps to reach either end
vector<int> minimumStepPolicy()
{
    vector<int> statePolicy(stateCount);
    int i = 1;

    while (i < statePolicy.size())
    {
        statePolicy[i] = min(i, numStates - i);
        i++;
    }

    statePolicy[0] = -1; // Exit state
    statePolicy[numStates] = -1; // Exit state

    return statePolicy;
}
```

```cpp
// Define a policy that always selects a fixed step size
vector<int> fixedStepPolicy()
{
    vector<int> statePolicy(stateCount, 1);
    statePolicy[0] = -1; // Exit state
    statePolicy[numStates] = -1; // Exit state
    return statePolicy;
}

// Check if the difference between two utility vectors is below the threshold
bool convergenceCheck(vector<double> oldUtility, vector<double> newUtility)
{
    double squareDifference = 0;
    int i = 0;

    while (i < oldUtility.size())
    {
        squareDifference += pow(oldUtility[i] - newUtility[i], 2);
        i++;
    }

    squareDifference = sqrt(squareDifference);
    return squareDifference < epsilon;
}
```

```cpp
// Calculate the utility vector for a given policy
vector<double> policyUtility(vector<int> statePolicy)
{
    vector<double> oldUtility(stateCount, 0);
    vector<double> newUtility(stateCount, 0);
    oldUtility[numStates] = 2 * numStates;
    newUtility[numStates] = 2 * numStates;

    bool converged = false;

    while (!converged)
    {
        oldUtility = newUtility;
        converged = true;

        for (int i = 1; i < statePolicy.size() - 1; i++)
        {
            newUtility[i] = headProbability * (discountFactor * oldUtility[i + statePolicy[i]]) + (1 - headProbability) * (discountFactor * oldUtility[i - statePolicy[i]]

            // Check for convergence in this state
            if (abs(newUtility[i] - oldUtility[i]) >= epsilon)
            {
                converged = false;
            }
        }
    }

    return newUtility;
}
```

```cpp
// Calculate the optimal policy using value iteration
vector<int> optimalValueIterationPolicy()
{
    vector<double> oldUtility(stateCount, 0);
    vector<double> utility(stateCount, 0);
    vector<int> policy(stateCount, 0);

    do
    {
        oldUtility = utility;
        for (int j = 0; j < stateCount; j++)
        {
            if (j == numStates)
            {
                utility[j] = 2 * numStates;
            }
            else if (j == 0)
            {
                utility[j] = 0;
            }
            else
            {
                double maxUtility = -1e37;
                for (int i = 0; i <= min(j, numStates - j); i++)
                {
                    if (maxUtility < (headProbability * (discountFactor * oldUtility[i + j]) + (1 - headProbability) * (discountFactor * oldUtility[j - i])))
                    {
                        policy[j] = i;
                        maxUtility = headProbability * (discountFactor * oldUtility[i + j]) + (1 - headProbability) * (discountFactor * oldUtility[j - i]);
                        utility[j] = maxUtility;
                    }
                }
            }
        }
    } while (!convergenceCheck(utility, oldUtility));

    return policy;
}
```

_ASSIGN_3A_GROUP_21.cpp > main()

```cpp
// Calculate the optimal policy using policy iteration
vector<int> optimalPolicyIterationPolicy()
{
    vector<double> utility(stateCount, 0);
    vector<int> policy(stateCount, 0);
    vector<int> previousPolicy(stateCount, 0);

    do
    {
        previousPolicy = policy;
        for (int j = 0; j < stateCount; j++)
        {
            double maxUtility = policyUtility(policy)[j];
            for (int i = 0; i <= min(j, numStates - j); i++)
            {
                double actionUtility = (headProbability * (discountFactor * policyUtility(policy)[i + j]) + (1 - headProbability) * (discountFactor * policyUtility(policy)[j - i]));
                if (maxUtility < actionUtility)
                {
                    maxUtility = actionUtility;
                    policy[j] = i;
                }
            }
        }
    } while (!(previousPolicy == policy));

    return policy;
}

int main()
{
    cout << "Using Minimum Step Policy:" << endl;
    cout << "State | Action | Utility" << endl;
    vector<int> minimumStepPolicyResult = minimumStepPolicy();
    vector<double> minimumStepPolicyUtility = policyUtility(minimumStepPolicyResult);
    for (int x = 0; x < minimumStepPolicyResult.size(); x++)
    {
        cout << x << " " << minimumStepPolicyResult[x] << " " << minimumStepPolicyUtility[x] << endl;
    }

    cout << endl;
    cout << "Using Fixed Step Policy:" << endl;
    cout << "State | Action | Utility" << endl;
    vector<int> fixedStepPolicyResult = fixedStepPolicy();
    vector<double> fixedStepPolicyUtility = policyUtility(fixedStepPolicyResult);
    for (int x = 0; x < fixedStepPolicyResult.size(); x++)
    {
```

_ASSIGN_3A_GROUP_21.cpp > main()

```cpp
    {
        cout << x << " " << fixedStepPolicyResult[x] << " " << fixedStepPolicyUtility[x] << endl;
    }

    cout << endl;
    cout << "Optimal Policy using Value Iteration:" << endl;
    vector<int> optimalValueIterationResult = optimalValueIterationPolicy();
    vector<double> optimalValueIterationUtility = policyUtility(optimalValueIterationResult);
    for (int x = 0; x < optimalValueIterationResult.size(); x++)
    {
        if (x == numStates)
        {
            cout << x << " -1 " << optimalValueIterationUtility[x] << endl;
        }
        else if (x == 0)
        {
            cout << x << " -1 " << optimalValueIterationUtility[x] << endl;
        }
        else
        {
            cout << x << " " << optimalValueIterationResult[x] << " " << optimalValueIterationUtility[x] << endl;
        }
    }

    cout << endl;
    cout << "Optimal Policy Using Policy Iteration:" << endl;
    vector<int> optimalPolicyIterationResult = optimalPolicyIterationPolicy();
    vector<double> optimalPolicyIterationUtility = policyUtility(optimalPolicyIterationResult);
    for (int x = 0; x < optimalPolicyIterationResult.size(); x++)
    {
        if (x == numStates)
        {
            cout << x << " -1 " << optimalPolicyIterationUtility[x] << endl;
        }
        else if (x == 0)
        {
            cout << x << " -1 " << optimalPolicyIterationUtility[x] << endl;
        }
        else
        {
            cout << x << " " << optimalPolicyIterationResult[x] << " " << optimalPolicyIterationUtility[x] << endl;
        }
    }

    return 1;
}
```

We observed that the optimal policy found by both value iteration and policy iteration algorithms was consistent.

The optimal policy maximizes the expected return while taking into account the probability of winning or losing.

Experimenting with different values of p, N, and γ can impact the optimal policy.

In some cases, the transition cost of -1 for non-terminal states can lead to different optimal policies.

Conclusion:

- In this work, we investigated several rules and methods for resolving the gambler's dilemma.
- To identify the best policy, value iteration and policy iteration techniques were also used.
- The policies and parameters you choose, such p, N, and, can have a big impact on the outcomes.
- It is advised to do further trials with various factors to determine how they affect the best course of action.

<u>THANK YOU</u>