CS747 Foundations of Intelligent and Learning Agents

Multi-Armed Bandits

Assignment 2

Param Rathour | 190070049

Autumn Semester 2021-22

Contents

0 Introduction

In this assignment, I have written code to compute an optimal policy for a given MDP using Value Iteration, Howard's Policy Iteration, and Linear Programming.

Then, I used this solver to find an optimal policy for each player in the game of Anti-Tic-Tac-Toe.

1 Task 1 - Implementation of Fundamental Algorithms

These algorithms are implemented by `planner.py`. `transitions` contains next state, rewards, and probabilities for each state-action pair. In this way, we can save memory compared to other approach of have rewards and probabilities for each state-action-next state pair.

1.1 Value Iteration

The algorithm is done exactly as discussed in class.

```
def valueIteration(numStates, numActions, endStates, transitions, mdpType, discount):
    values = np.zeros(numStates)
    newValues = np.zeros(numStates)
    actions = np.zeros(numStates, dtype = int)
    while True:
        for s1 in range(numStates):
            if not endStates[s1]:
                temp = np.zeros(numActions)
                for ac in range(numActions):
                    data = transitions[s1][ac]
                    temp[ac] = sum([data[i][2]*(data[i][1]+discount*values[data[i][0]])
                                    for i in range(len(data))])
                actions[s1] = np.argmax(temp)
                newValues[s1] = temp[actions[s1]]
        if np.linalg.norm(values - newValues, ord = np.inf) < 1e-10:
            break
        else:
            values = [newValues[i] for i in range(numStates)]
    values = [newValues[i] for i in range(numStates)]
    return values, actions
```

1.2 Howard's Policy Iteration

The algorithm is done exactly as discussed in class. The assumption I made while choosing action from improvable states was to always select the lowest action (lowest index). This gave the advantage of skipping the calculation of action value functions for other actions at those state. Hence speeding up the algorithm.

```
def howardPolicyIteration(numStates, numActions, endStates, transitions, mdpType, discount):
    actions = np.zeros(numStates, dtype = int)
    improvableStates = True
    while improvableStates:
        improvableStates = False
        values = np.zeros(numStates)
        newValues = np.zeros(numStates)
        while True:
            for s1 in range(numStates):
                if not endStates[s1]:
                    data = transitions[s1][actions[s1]]
                    newValues[s1] = sum([data[i][2]*(data[i][1]+discount*values[data[i][0]])
                                        for i in range(len(data))])
            if np.linalg.norm(values - newValues, ord = np.inf) < 1e-10:
                break
            else:
                values = [newValues[i] for i in range(numStates)]
        for s1 in range(numStates):
            for ac in range(numActions):
                data = transitions[s1][ac]
                actionValues = sum([data[i][2]*(data[i][1]+discount*values[data[i][0]])
                                   for i in range(len(data))])
                if actionValues > values[s1] and (actionValues - values[s1] > 1e-7):
                    improvableStates = True
                    values[s1] = actionValues
                    actions[s1] = ac
                    break
    return values, actions
```

1.3 Linear Programming

The algorithm is done exactly as discussed in class.

```
def linearProgramming(numStates, numActions, endStates, transitions, mdpType, discount):
    problem = pulp.LpProblem("Values", pulp.LpMinimize)
    decisionVariables = [pulp.LpVariable('V('+ str(state) +')') for state in range(numStates)]
    cost = sum(decisionVariables)
    problem += cost

    for s1 in range(numStates):
        for ac in range(numActions):
            data = transitions[s1][ac]
            problem += (decisionVariables[s1] >=
                        pulp.lpSum([data[i][2]*(data[i][1]+discount*decisionVariables[data[i][0]])
                                   for i in range(len(data))]))

    problem.solve(pulp.PULP_CBC_CMD(msg = None))
    values = np.zeros(numStates)
    for value in problem.variables():
        values[int(value.name[2:-1])] = value.varValue
    actions = np.zeros(numStates, dtype = int)
    for s1 in range(numStates):
        actionValues = np.zeros(numActions, dtype = int)
        for ac in range(numActions):
            data = transitions[s1][ac]
            actionValues[ac] = sum([data[i][2]*(data[i][1]+discount*values[data[i][0]])
                                    for i in range(len(data))])
        actions[s1] = np.argmin([abs(values[s1] - i) for i in actionValues])
    return values, actions
```

2 Task 2 - Anti Tic Tac Toe Fixed Policy

Here, I created `encoder.py` and `decoder.py`. With this the ATTT policy is converted into a MDP with a player as agent and another as environment. For this, I defined transistion parameters between states, actions.

I am using Value Iteration as my default algorithm, and I made a slight modification to accomodate the fact that all state action pairs might not be defined. To see if they exist before using them. This `try` approach is also used in encoder extensively.

```
try:
    data = transitions[s1][ac]
except:
    continue
```

Unfortunately, only encoder worked and hence, this task remains incomplete.

3 Task 3

I think the sequence of policies generated for each player are guaranteed to converge. The major reason is that number of states, actions are finite. So, with our prcoess, the value function of each player increases and will ultimately reach the optimal value.

4 Conclusion

It was a pleasure doing this assignment. I learnt a lot about MDPs and Python overall. Thanks.