MDL Assignment-2 Part 3 REPORT

Team Number : 22

Team Name : Room543

Mihir Bani, 2019113003

Amul Agrawal, 2019101113

Part-3: Linear Programming

Overview

In LP, a linear function is maximized/minimized with some given constraints. This can also be used to solve MDPs. Then we can create an optimized policy based on the solutions of LP model.

Let,

- n_a = total number of actions across all states, and
- n_s = total number of states

Then we define - $< A, R, x, \alpha >$

- A is a matrix that stores all the probabilities values of moving in or moving out of a given state. Its dimensions are $(n_s \times n_a)$.
- R is the reward for taking an action from a state. Its dimensions are $(1 \times n_a)$.
- x stores the number of times a particular action should be taken from state s. Its dimensions are $(n_a \times 1)$.
- α denotes the probability distribution of each state in starting of the MDP. Its dimensions are$(n_s \times 1)$.

Now, to solve a MDP to get an optimum policy π^* with LP. It tries to pick the best action that is possible from the current state.

> ### Solving MDP with LP:

📌 Objective: **Maximize** Rx

This is the final Reward value, calculated by multiplying the rewards with the number of actions taken. The main objective is to maximize our reward.

📌 Constraints: $Ax = \alpha, x \geq 0$

The product of A and x, should be equal to the initial probability distribution of each state as in the starting of MDP.

Each value of $x \geq 0$, as it denotes the number of times a action was taken. So its a nonpositive number.

Making A array

A matrix denotes the probability of going out or going into a state, by taking a particular action. And so it is a $(n_s \times n_a)$ matrix.

In our case, a state is defined as a tuple:

`s : (position_ij, materials, arrows, state_mm, health_mm)`

So the total number of states

`n_s = MAX_POSITIONS * (MAX_MATERIALS + 1) * (MAX_ARROWS + 1) * MAX_MM_STATES * NUM_MM_HEALTHS`

For total number of actions, we scan through all the states and then count the number of actions that can be taken from that state, and add all of them.

```
# PSEUDO CODE TO FIND NUMBER OF ACTIONS
def get_dimensions():
    dim = 0
    Loop for state in all states:
        if state.actions() is None:
            dim += 1
        else:
            dim += len(state.actions())
     return dim
```

Let A_{ij} denote a value of A at index i, j. (State: i, Action: j)

It stores the probability of action j taken at the state i. This value is positive when the action helps in moving out of the state, and negative for moving into the state.

$$0 \leq |A_{ij}| \leq 1, for\ i \leq n_s\ ,\ j \leq n_a$$

If the action j is a NOOP action, i.e., it is an end action in a state. Only the probability is added to one state, without a negative corresponding value in other state.

Storing values in the A matrix :

- A is initiated as a zero matrix first. Then iterate through each state, i.e. row wise in the matrix ***A*.**
- For each state, we find all the actions that are possible.
- If any actions are not possible, the column is incremented without updating any value. This is the NOOP action, and `A[state,j] = 1`, which denotes the probability of moving out of state is **1**.
- Otherwise, for each action, we add the probability to A[state, j] and the negative of this probability to the index of the state reached after taking the action.

  `A[state, j] += p` and `A[next_state, j] += -p`

```
# PSEUDO CODE FOR MAKING MATRIX "a"
def get_a():
    a = np.zeros((NUM_STATES, NUM_ACTIONS), dtype=np.float64)
    cnt = 0
    for state in STATES:
        actions = state.actions()

        if actions is None:
            self.action_at_idx.append("NONE")
            a[i][cnt] += 1
            cnt += 1
            continue

         for action, results in actions.items():
             self.action_at_idx.append(action)
             for result in results:
                 p = result[0]
                 next_state = result[1]
                 a[i][cnt] += p
                 a[State(*next_state).get_index()][cnt] -= p

             check if a[i][cnt] <= 1.005 # considering floating point errors
             # increment cnt
             cnt += 1
    return a
```

Finding the optimal policy

Once we solve the LP problem, by maximizing Rx , which is our total reward owing to some constraints. We find the x matrix/vector, and this stores the number of times an action should be taken at any state.

$$max(Rx) \mid Ax = \alpha,\ x \geq 0$$

We use this x values to find which action should be taken the highest number of times at a particular state, and then this action should be selected for the optimal policy.

Procedure:

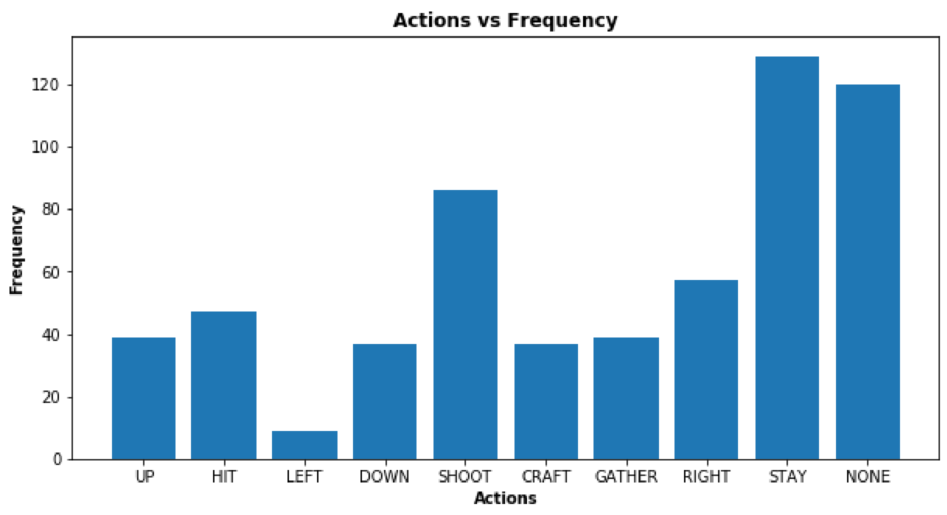
- We iterate through all the states, and for each state, find the set of actions available.
- If no action is available, that means its a NOOP action.

- Otherwise, we get an array of possible actions from a state, and their optimum frequency is stored in x vector.
- We find the action for which the value in x is maximum, from the set of actions for the chosen state.
- This `(state, action)` pair is stored in the policy.

```
# PSEUDO CODE FOR FINDING THE POLICY
    def get_policy():
        idx = 0
        for state in STATES:
            actions = state.actions()
            if actions is None:
                idx += 1
                best_action = "NONE"
                policy.append([state, best_action])
                continue

            act_idx = np.argmax(x[idx : idx+len(actions.keys())])
            idx += len(actions.keys())
            best_action = list(actions.keys())[act_idx]
            policy.append([state, best_action])
```

Analysis



1. **Centre Square:**
    1. If MM is Dormant:
        1. if arrows ≥ 2:
           He moves RIGHT to the east square, where he can SHOOT with much higher probability
        2. if arrows > 0:
           He SHOOTs from the same square.

      3. if arrows = 0:
         1. if materials > 0:
            He moves UP, where he can CRAFT arrows.
         2. else:
            He HITs with blade.
   2. If MM is Ready:
      1. Materials > 0 and Arrows < 3 and Materials > Arrows:
         He moves UP to North square, so he can make arrows or stay.
      2. else:
         He moves DOWN, to gather more materials, or stay.
2. **East Square**
   1. if Arrows = 0 or MM_health = 100:
      He tries HIT action.
   2. Otherwise,
      He SHOOTs all his arrows first.
3. **North Square**

   Takes the first possible action in the following sequence.
   1. Materials > 0 and Arrows < 3 or Materials > Arrows:
      He CRAFTs arrows.
   2. MM is Ready:
      when no more arrows can be crafted, he STAYs here.
   3. MM is Dormant:

      He moves DOWN, to Center square, where he can shoot MM.
4. **South Square**
   1. MM is Ready:
      He GATHERs, as STAY as no productive use.
   2. MM is Dormant:
      1. if Materials = max:
         He prefers going UP.
      2. if materials ≠ 0:
         He STAYs.
5. **West Square**
   1. MM is Dormant:

    1. if MM_health < 50:
       He STAYs.

    2. else:
       He moves RIGHT to Center square.

2. MM is Ready:

    1. if MM_health < 75:
       He chooses to STAY.

    2. else:
       He SHOOTs arrows.

Multiple policies : Why? What changes in code to generate another policy?

Yes there can be multiple policies that are optimal, depending on the matrices A, α, R , etc.

- For a given A, α, R ; we can get multiple policies on solving LP, as x can have the same value for more than one action, that can be taken from each state. So there is a choice of choosing one action in the plan. In our code, we used `argmax` function from Numpy. This finds out the index of the first occurrence of the max element. If we use something else like, find all the max elements (if more than one) and then choose randomly from this set, we will get a different optimal policy.
- Changing the matrix A will surely change the policy. As A contains the sets of actions and their probabilities of happening for each state. As the probability is changed, the expected utility will also change for each state.
- If we change the initial probability distribution α of starting from a state, then this changes the constraints of the LP, and so the policy will surely change.
- Changing Rewards and Step Cost in the vector R, will also affect the policy a lot. A higher reward for some action means it will be favoured more, and a higher Step Cost means that the policy will try to minimize the number of steps.