LinearProgramming

April 10, 2021

1 Linear Programming

Tanishq Goel : 2019114015
Rajat Kataria : 2019101020

1.1 Part 3

1.1.1 Libraries Used

- numpy, for arrays and mathematical functions
- cvxpy, for creating and solving linear programming model
- os and json, for dumping the final dictionary as a json object

1.1.2 Key for Matrix A

For constructing the matrix A, **key** was made to hash all the states.
The key is as follows:

0 : ['C', 0, 0, 'R', 0]
1 : ['C', 0, 0, 'R', 1]
2 : ['C', 0, 0, 'R', 2]
3 : ['C', 0, 0, 'R', 3]
4 : ['C', 0, 0, 'R', 4]
5 : ['C', 0, 0, 'D', 0]
6 : ['C', 0, 0, 'D', 1]
7 : ['C', 0, 0, 'D', 2]
8 : ['C', 0, 0, 'D', 3]
9 : ['C', 0, 0, 'D', 4]
.
.
.
590 : ['W', 2, 3, 'R', 0]
591 : ['W', 2, 3, 'R', 1]
592 : ['W', 2, 3, 'R', 2]

593 : ['W', 2, 3, 'R', 3]
594 : ['W', 2, 3, 'R', 4]
595 : ['W', 2, 3, 'D', 0]
596 : ['W', 2, 3, 'D', 1]
597 : ['W', 2, 3, 'D', 2]
598 : ['W', 2, 3, 'D', 3]
599 : ['W', 2, 3, 'D', 4]
Function get_hash() is used to initialize the key. Dimensions of the matrix A is 600 X 1936.

1.1.3 Creating Matrix A

The A matrix is very important for solving the linear programming problem given to us. Therefore, it needs to be made with utmost care and caution.

The procedure is the following: `- a = np.zeros((NUM_STATES, self.dim), dtype=np.float64)`

- We initialize matrix with 0.0 with the dimensions 600,1936
- No. of columns comes out to 1936 which is calculated by function get_dimension()
- Then we have to iterate through all the possible states from 1 to 600. The same will function as our rows for the A matrix.
- Next we need to consider the valid state action pairs. We remove the invalid state action pairs. For example shooting when there are no arrows left. Function is_action_valid() is used to check for valid actions and all of the mentioned conditions were taken into account.
- If the action is valid then we add +1 to its corresponding place in A matrix as it definitely goes out of this state irrespective of whether it will come back or go to some other state.
- Then we try to find the next possible states according to the action chosen
- Then we subtract the probability values from the corresponding place in A matrix as this action depicts an inflow now which should always be subtracted.
- Then, we increment the column number and move to next combination of state and valid state-action pairs.
- Note that we also have to take care of our 'final' state, i.e., when the monster's health reaches 0. Such a state will have only one "NONE" action and to depict the same, we just put 1 in its corresponding place in the A matrix.

1.1.4 Procedure for finding the policy

- For finding the policy, the **X__Matrix** was returned as solution from cvxpy and the **key** was used.
- Now each element of the **X__Matrix** corresponds to a pair of .
- Now the max of x corresponding to same state will be taken to find the optimal policy/action.
- So policy list is calculated using function **get__policy()** which is explained as follows:
- Firstly, we take only the values of the x array and append it to a new list called 'xarr'. The lenght of this array is 1936, which is the number of valid state action pairs. Therefore, this array contains the utility values of carrying out a certain valid action in a certain state.
- Then, we traverse through all the possible states. Since finding the policy means finding the best action for all the possibly 600 states, we will have to iterate through them all.
- Then we consider all the possible actions of the state according to the position the state is in, i.e., center, east, west, etc.
- Then we initialize the maximum utility value for the state as the starting action value. Then we iterate through all the actions and try to find the maximum utility value. We store the action name along with the value.
- Then we append the value and the name of the action to a temporary list, 'temp'. Before appending we make sure to multiply by state health with 25 as our initial health array was [0,1,2,3,4] instead of [0,25,50,75,100].
- Afterwards, we append the entire list 'temp' into our 'policy' list.
- Likewise we carry out the same steps for all the rest of the states.
- Note that we will have to take of our terminal state, i.e., when the monster health reaches 0, especially again. For this we append the special action "NONE" without much thought as it is the only possible action that can be used to denote taking the flow out of the final state.

1.1.5 Analyzing the results based on policies

- UP: 39times ; DOWN: 38times ; RIGHT: 58times; LEFT: 9 times; SHOOT: 85times; STAY: 127times; HIT:47 times , GATHER:39 times , CRAFT: .38times
- The most chosen action in general was the STAY action. It was the best action for about 127 of all possible states. NONE follows because of obvious reasons
- Most preferred way to attack was SHOOT which corresponds to twice as many states as that of HIT

- The next most chosen action was RIGHT(around 60), so IJ getting closer to the monster being a good move is also hinted by this policy.
- The actions UP, DOWN, GATHER, CRAFT all occurred in almost the same window of around 40 times. They were also important to the policy but a mere 40ish times out of 600.
- The least common action was LEFT as it occured only 9 times out of 600. This also goes on to show important behavior of IJ in the policy. As there is no reward, IJ wants to defeat MM in the least number of steps so as to get lower step cost.

The above data helps us to analyze the policy.This also goes on to show important behavior of IJ in the policy. As there is no reward, IJ wants to defeat MM in the least number of steps so as to get lower step cost. If there would have been some reward then IJ would have preferred to stay in NORTH or SOUTH squares when MM is in READY state so as to averse the risk but as there is no reward, IJ moves hastily and tries to kill MM as soon as possible.

2 Multiple policies

Yes, it can be due to **commutability** . This phenomenon occurs when two or more actions are equally beneficial at a given state, and therefore, their inherent order doesn't change the objective values,but results in the existence of multiple equally desirable policies. - If the x values in the solutionx array for the actions for a state are the same, it is one's choice to choose which action for that state.
- Now this choice depends on the order of the actions.If two actions have the same x value for a state and are max among all x for that state, one can choose any one of the two actions. My algorithm chooses the action that lies at a lesser index in the order of actions.
- Depending on the order of actions the A matrix would also change as each column in A matrix basically represents a pair and for a fixed state , the actions goes in the order fixed earlier. So if the order is changed, the only difference that will happen in A matrix would be that the columns would shift a few indexes backward or forward.
- Changing the order of actions would not change the r matrix or the alpha as it is dependent only on the state and reward for each state and not on actions on a state.
- Changing the order of action would change the solutionx matrix as each column also corresponds to the pair.