Assignment-2 Part-3
Report

Akshit Garg (2018113006)
Jashn Arora (2018114006)

**** The order of actions followed in the assignment everywhere is:

- **1: NOOP 2: SHOOT 3: DODGE 4: RECHARGE**

❖ Procedure for making the A matrix

➢ For constructing the A matrix , A dictionary named **'key'** was constructed.So first let us understand the construction of 'key' dictionary.

➢ Key was initialised as follows:

- Key = {
  '000': [0,[ ],[ ],[ ],[ ]],
  '001': [1,[ ], [ ] ,[ ], [ ]],
  '002': [2,[ ],[ ],[ ],[ ]],
  '010': [3,[ ],[ ],[ ],[ ]],
  '011': [4,[ ],[ ],[ ],[ ]],
  .
  .
  .
  .
  .
  '430': [57,[ ],[ ],[ ],[ ]]
  '431': [58,[ ],[ ],[ ],[ ]]
  '432': [59,[ ],[ ],[ ],[ ]]
  }

➢ Each entry in the dictionary as follows:

- '<State Representation>': ['<state count>' , [ ] , [ ] , [ ] , [ ]]
- Each list in each element of the key dictionary represents action for that state.
- For Example:
  - '001' : [1,[list for noop] , [list for shoot] , [list for dodge]], [list for recharge]]]
- Each action list in each element of the key will further consist of lists each of them consisting of pairs like '<state representation of the target state on taking that action>, <probability>'.

- An example an action list for SHOOT may be:
  - [[‘230’,0.8],[‘231’:0.2]]
  - This represents that, shoot action on this state can lead to state ‘230’ with prob. 0.8 and state ‘231’ with prob. 0.2.
- An example of an element in the key dictionary can be:
  - '111': [16, [ ], [['000', 0.5], ['100', 0.5]], [['120', 0.8], ['110', 0.2]], [['112', 0.8]]]
  - This entry represents the following:
    - State ‘111’ has index 16
    - From this state NOOP action can not be performed as an action list corresponding to NOOP action is empty.
    - Taking SHOOT action on state ‘111’ leads to state ‘000’ with prob. 0.5 and ‘100’ with prob 0.5 .
    - Taking DODGE action on state ‘111’ leads to state ‘120’ with prob. 0.8 and ‘110’ with prob 0.2.
    - Taking RECHARGE action on state ‘111’ leads to state ‘112’ with prob. 0.8 .
- The function **construct_key()** initialises the key matrix.
- The function **fill_key()** loops over all states and fills each of its elements by calling functions **shoot(), recharge(), noop(), dodge()** which fills the action list corresponding to the actions for every state.

➢ Now the function **fill_A_matrix()** constructs the matrix of size 60*100 i.e <no.states>*<sum of count of actions that can be performed at each state> and is initialised with 0 for each element of the matrix .

➢ **fill_A_matrix()** runs as follows:

- It loops over each element of the **key** dictionary and for each action list for that element that has its size greater than zero corresponds to each column of the A matrix.
- So an element of a key having 2 of its action list with size greater than 0 corresponds to 2 columns in A matrix for the state represented by the element of the key.
- The row index in A matrix for a state is analogous to the state number represented as the first element of each element of the key dictionary. For example:
  - '111': [16, [ ], [['000', 0.5], ['100', 0.5]], [['120', 0.8], ['110', 0.2]], [['112', 0.8]]]
  - Row index of state ‘111’ will be 16 in A matrix.
- Now let us try building A matrix for one element of the key dictionary.Let us consider the following element.
  - '111': [16, [ ], [['000', 0.5], ['100', 0.5]], [['120', 0.8], ['110', 0.2]], [['112', 0.8]]]

- Now as the NOOP action list is empty for this state, NOOP action would not represent a column in A matrix for this state.
- As the SHOOT list is not empty it will represent a column in A matrix .Now consider each pair of 'SHOOT' the list for this state and fill the column corresponding to SHOOT action at this state.
  - For pair ['000', 0.5] , the value of the row corresponding to state '000' and the column for SHOOT of the current state'111' is subtracted with 0.5 marking the influx to state '000' and the value corresponding to the state '111' in the row and the column corresponding to SHOOT of the current state '111' is added with 0.5 marking the outflux from state '111'.
  - Similarly for pair ['100', 0.5] , the value of the row corresponding to state '100' and the column for SHOOT of the current state'111' is subtracted with 0.5 marking the influx to state '100' and the value corresponding to the state '111' in the row and the column corresponding to SHOOT of the current state '111' is added with 0.5 marking the outflux from state '111'.
- Similar things are done for DODGE and RECHARGE.

○ **NOTE**** For terminal states having only the NOOP action self loop, only the outflux was taken into consideration and not the influx.

➢ So this is how the A matrix was constructed.

❖ Procedure for finding the policy

➢ For finding the policy, the **solutionx** matrix of size 100*1 which was returned as a solution from cvxpy and the **key** dictionary was used.

➢ Now each element of the 'solutionx' matrix corresponds to a pair <state,action>.

➢ Now the max of x corresponding to the same state should be taken and the action which it corresponds to will be the optimal action for that state.

➢ So the policy list is calculated using **get_policy()** function which goes as follows:

- First of all there is a variable **count** that marks the index till which we have traversed in solutionx matrix.It is initially placed at zero.
- For each element in the key dictionary , it forms a list containing action no. all actions that have their action list size greater than 0 as they would have an entry in solutionx matrix for that state. For example:
  - If a state has 3 action lists of size greater than zero corresponding to SHOOT , DODGE and RECHARGE. So the list would be [2,3,4].
  - If a state has 2 action lists of size greater than zero corresponding to SHOOT , DODGE. So the list would be [2,3].

        - If a state has 2 action lists of size greater than zero corresponding to SHOOT and RECHARGE. So the list would be [2,4].
        - If a state has 1 action list of size greater than zero corresponding to NOOP. So the list would be [0]
    - Now max. element is found from the index <**count**> to <**count** + **no. of actions possible at that state**> in the solutionx matrix. The action corresponding to this value in the list formed in the previous step is taken as the optimal action for the state.
    - This data is filled in the policy list in the required format.
    - Finally the '**count'** is updated to '**count** + **no. of actions possible at that state'**

❖ Can there be 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.

- The order I chose was:
    - **1: NOOP 2: SHOOT 3: DODGE 4: RECHARGE**
- 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 <state,action> 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 <state,action> pair.