Machine, Data and Learning

Assignment 2 Part 3

Vasu Singhal - 2018101074
Tanish Lad - 2018114005
Team 43

Report

Brief Explanation of what each method of the class does:

- **__init__**

  Initializes all the variables and arrays required for the execution of rest of the code. It calls the solve_lp() method.

  Our team number is **43**. The penalty is **-10**.

- **solve_lp**

  It calls all the below methods one by one.

- **initialize_states**

  This method forms a list of tuples of all the possible states in the form of (h, a, s) where h represents the enemy health, a represents the number of arrows remaining and s represents the stamina.

- **initialize_possible_actions**

  This method determines the list of all possible actions that Lero can perform when it is present in that state.. This data is stored in the form of a dictionary, where the key is the state tuple and the value is the list of all possible actions that Lero can perform when it is present in that state.

  NOTE: The order of the actions we chose is **(“NOOP”, “RECHARGE”, “DODGE”, “SHOOT”)**. The A matrix will vary depending on the choice of the order of actions taken.

- **flow**

  This method takes two arguments: **initial_state** and **final_state**. It then iterates over all the possible actions of **initial_state** and finds with what probability, on taking a particular action in **initial_state**, takes it to the **final_state** or away from the **final_state**. This is used in building the **A** matrix.

- **initialize_Amatrix**

  This method forms the A matrix, which is used in formulating the linear program. The procedure for building this A matrix is given in detail later.

- **initialize_reward**

  This method forms the reward ‘R’ array. Reward (Step cost) is 0 for that state when the enemy health is 0, else the reward is -10.

- **initialize_alpha**

  This method calculates alpha (the initial probability distribution). Initially, Lero has 3 arrows, max stamina, and the MD has full health. So probability of the state tuple (4, 3, 2) is 1, and for the other states it is 0.

- **solve**

  This method solves the Linear Program and stores the value of the objective and the optimized x array.

  The given situation is posed as a LP Problem in the following way:

  Maximize

  r.x

  On the constraints

  1. **A.x = alpha**
  2. **x >= 0**

  ‘**.**’ represents matrix multiplication. The x array represents the total expected number of times state i is visited.

- **get_policy**

  This method calculates the policy given the x array. It finds the best action that should be performed if Lero is in that state. It is calculated by finding the best value in the subarray representing that state and the action that gives the best value is our optimal action.

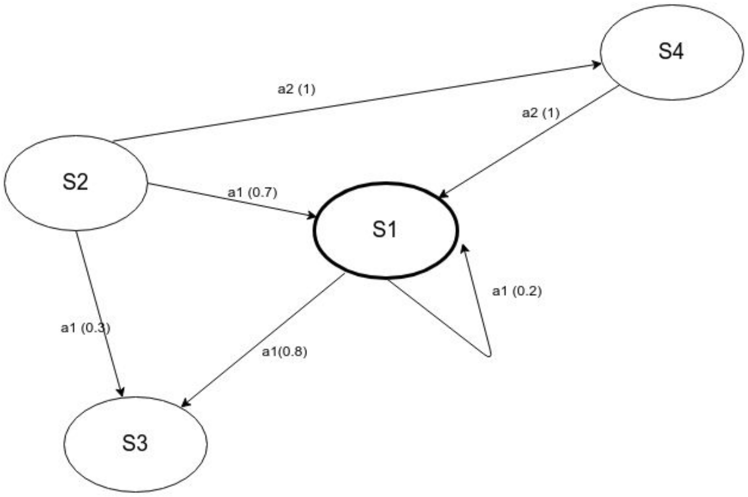
- **dump_to_file**

  This method forms the dictionary according to the format specified in the assignment. It then creates a ‘outputs’ directory and dumps the dictionary in json format in the outputs directory. The file is named output.json.

Procedure for making the A matrix

We will explain our procedure for making the A matrix through an example.

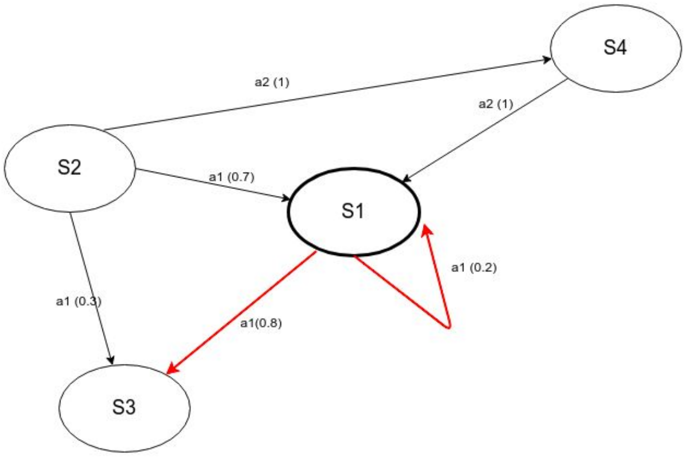
- On the next page -



Suppose we need to build the A matrix for the above states, and we are currently building the list for state s1.

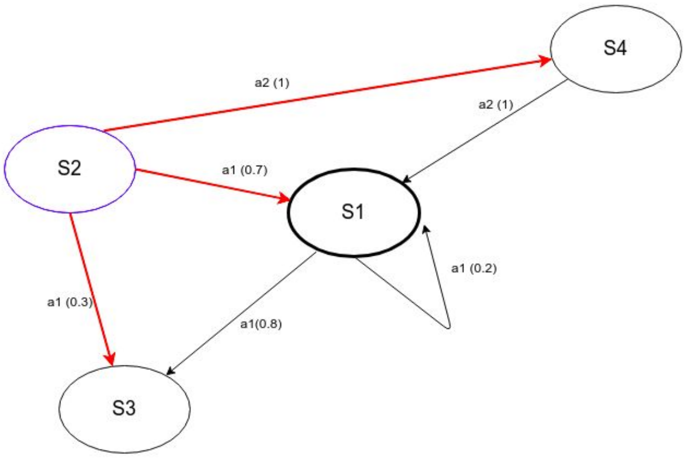
The list of all the actions is [S11, S21, S22, S30, S42]. (S30 is the NOOP action, not shown in the diagram)

For **building the list for S1**, we go over all the states (including S1) and add the contribution of their actions.



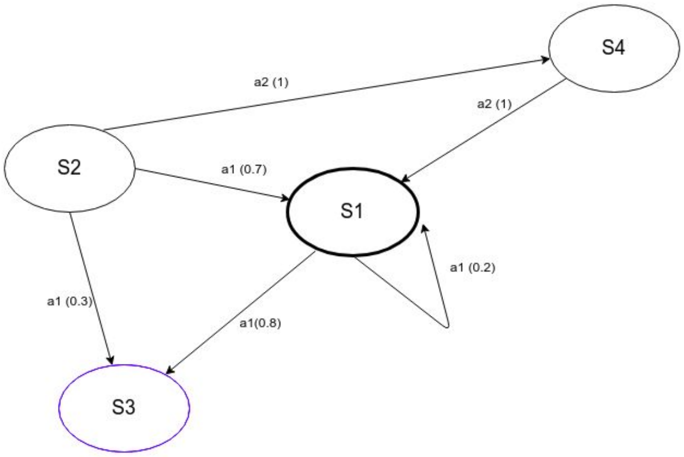
The first state that we visit is S1. It can perform only one action a1, which takes it out from s1 with prob 0.8. Thus we append 0.8 in the list of S1. The self loop is not considered because considering it will make the lpp unbounded.

So the list becomes [0.8]



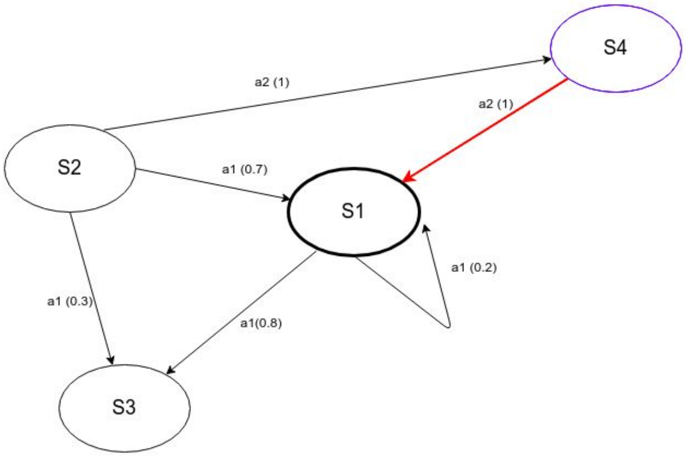
Next we visit state S2, which can perform two actions a1 and a2. Action a1 takes state S2 to S1 with probability 0.7. So we append -0.7. Action a2 takes S2 to S1 with 0 probability. So we also append 0. Note that we are only building the list of S1 here.

Thus the list becomes (0.8, -0.7, 0)



Next we visit state S3, which can only perform NOOP action, which takes it to state S1 with 0 probability. Thus we append 0.

Thus the list becomes (0.8, -0.7, 0, 0)



And lastly we visit state S4, which can perform only one action a2. This action takes S4 to S1 with probability 1. So we append -1.

Thus the list becomes (0.8, -0.7, 0, 0, -1)

Following a similar procedure, we build the list for all the remaining states, and then we append all these lists in a new list. This final list of lists is the required A matrix.

A = [[0.8, -0.7, 0.0, 0.0, -1.0],

[0.0, 1.0, 1.0, 0.0, 0.0] ,

[-0.8, -0.3, 0.0, 1.0, 0.0] ,

[0.0, 0.0, -1.0, 0.0, 1.0]]

Procedure for finding the policy

We will explain the procedure for finding the policy through another example.

Consider the following scenario.

The list of states S = [S1, S2, S3, S4, S5, S6]

The possible Actions A = {a1, a2, a3}

The list of all possible actions for each state is [(a1, a2), a1, (a1, a2, a3), a1, a1, a1]

The variable x array is [(x_{11}, x_{12}), x_{21}, (x_{31}, x_{32}, x_{33}), $x_{4,1}$, x_{51}, x_{61}]

We have the A matrix and the initial probability distribution alpha. We form the constraints and the objective function of the LP using the matrices. On feeding the constraints and the objective function to the LP solver, suppose we get the optimised x array as:

[(0.03, 0.17), 0.1, (0, 0.1, 0.3), 0.05, 0.15, 0.7]. Each element of the array x_{ia} represents the expected number of times action a is taken in state i.

Then the policy is formed as follows:

For each state i, select action a which gives the max x_{ia} value among all possible actions a. **For state i, $a_{optimal}$ = argmax(x_{ia}) for all a.**

For the above x array:

- For state S1, a2 gives the max value (0.17)
- For state S2, there is only one possible action: a1
- For state S2, a3 gives the max value (0.3)
- For states s4, s5 and s6, there is only possible action: a1

So the policy list is [a2, a1, a3, a1, a1, a1].

Can there be multiple policies?

Yes, there can be multiple policies.

- One of the reasons is commutability. This phenomenon occurs when more than one action has the maximum benefit. This results in the existence of multiple desirable optimal policies.

- The chosen policy depends on **how ties are broken**. Out of the multiple maximums, any action can be chosen. The decision is random and can be changed by adding a very small penalty to one of the actions.
- There are many cases in which the **order of operations dont matter**. Suppose the enemy health goes to 0 by performing this set of actions: (SHOOT, RECHARGE, SHOOT). But the enemy health will still go down to 0 on changing the order of the first two actions. This set of actions would also be optimal: (RECHARGE, SHOOT, SHOOT). Both policies are equally good. The decision is random and can be changed by adding a very small penalty to one of the actions.
- Note that the **order of operations does matter in the case of DODGE**. It is beneficial to DODGE when the stamina is 50, rather than to DODGE when the stamina is 100. Because there is a chance that stamina goes down directly to zero from 100 when the action DODGE is performed. So, in this case the action of DODGE is always preferred to be performed when the stamina is 50 and less preferred to be performed when the stamina is 100.
- The optimal policy also depends on the **order (preference) of actions**. If the order of actions chosen is different, then the **A matrix would be different, the R matrix would be different, alpha would still remain the same, the objective value would be the same**. But the policy may differ if there exist multiple maximums. So changing the preference of actions may lead to a change in the chosen policy.