Applying Reinforcement Learning to play Sokoban

Srinivas Mukund Vadrev*

[1]Student ID: 202005966

x2020fvn@stfx.ca

In this work, we formalize the popular Japanese game Sokoban as an MDP and Q learning to solve the problem

1. Introduction

Reinforcement learning is the branch of machine learning that deals with algorithms that can learn based on reward. For instance a positive reward may encourage an agent to take certain actions in certain states while negative rewards on the other hand may discourage an agent against something or to quickly act to minimize the negative score it has been obtaining. Rewards can also be zero and based on the various reward signals the agent is obtaining in a particular state, it can formulate a mapping function called policy using which it can take optimal actions after training. During the initial training period, the agent may not know which is the best action unless it randomly explores various actions in various states. Once an agent has explored these various actions and observed the rewards, it forms an opinion about what action to take when it is in a certain state i.e which actions are the best in a given state. As the agent forms opinions about various actions, it is now able to exploit this knowledge to take the best actions for the states it observes. This phenomenon is referred to as exploitation. A good agent must strike the perfect balance between exploration exploitation. Eventually an agent gets good enough that it can take the correct trajectory of actions and obtain rewards in the right sequence of states.

Sokoban is a 2D role playing game and was developed in 1982 by Japanese game developer Hiroyuki Imabayashi. It consists of several maps with each containing a small room filled with boxes and certain colored tiles. When a box has been correctly moved to its marked tile, it becomes red. The objective of the player is to move all the boxes onto their respective marked tiles, thereby making them all red. The game finishes when the player has successfully moved the last box onto its tile.

2. Formalizing Sokoban as an MDP

A problem can be formalized into a reinforcement learning problem using four components i.e. agent, environment, reward and actions. The agent observes a state of the environment and takes an action which results in a change in the environment state. In other words, upon taking an action the environmental state transitions to the next state with some probability and may or may not result in a reward. For basic reinforcement learning problems, a complete model of this mapping is required. So a tabular structure is created which has a list of all the possible states, actions and next states. This type of reinforcement learning is called model based RL as opposed to model free RL and this data structure is referred to as the transition function. Similarly, we have to build a function that will map all our states into next states using the actions. Reinforcement learning

```
# w,w,w,w,w,p,f,x,w,w,b,f,b,w,w,x,f,f,w,w,w,w,w,w
# 0 = wall; 1 =floor; 3 =player; 2 = box; 5 =x; 4= boxCorrect
import numpy as np

map = np.array([
    [
      [0,0,0,0,0],
      [0,3,1,5,0],
      [0,2,1,2,0],
      [0,5,1,1,0],
      [0,0,0,0,0]
    ]
])
```

Figura 1. Representation of 5 x 5 map. Note this is for representation and is not shape = 1 and 25

algorithms use Dynamic programming methods to estimate values for actions. There are many methods to achieve this. But, broadly they can be classed into policy based methods and value based methods. In policy based methods we have to first evaluate the current policy values v* before being able to estimate values for corresponding actions q*. However, in value based methods we are able to directly estimate the values of actions without policy evaluation as innately values and policies are tied.

In our problem, since the game is an RPG game, our actions will be right, left, up and down. If the character is directly in behind a box and executes the function up, then the character will replace the position of the box and the box will have automatically transitioned up due to the velocity of the character. Using this methodology, the agent can move around and also move the boxes. To achieve this, we design a transition function that takes the initial position of the player and then shifts the player and the environment appropriately around, based on the action.

3. THE STATE

The state can be represented as a 5 x 5 Sokoban map. In our experiments, we decided to go with a simple 5 x 5 map to keep the transition table light enough while also allowing us to test complex scenarios.

The map consists of the following elements and can be represented appropriately by Python variables as seen below. Although the variable mp itself is a 5x5 matrix, to simplify things, we are flattening the map into a 1x25 matrix to keep things simple.

Map = np.array([[Wall, Wall, Wall,Wall,Wall], [Wall, Floor, FloorDot, Floor, Wall], [Wall, Player, BoxIncorrect, Floor, Wall], [Wall, BoxIncorrect, Wall, Floor, Wall], [Wall, FloorDot, Floor, Floor, Wall], [Wall, Wall, Wall,Wall,Wall]])

InitPos = (0,0)

Sheet1

	State					Action	Next state					Probability
Section 1	W	W	W	W	W	Down	W	W	W	W	W	90.00%
	W	P	F	X	W		W	F	F	X	W	
	W	B	F	B	W		W	P	F	B	W	
	W	X	F	F	W		W	BC	F	F	W	
	W	W	W	W	W		W	W	W	W	W	
	W	W	W	W	W	Right	W	W	W	W	W	
	W	P	F	X	W		W	F	P	X	W	3.33%
	W	B	F	B	W		W	B	F	B	W	
	W	X	F	F	W		W	X	F	F	W	
	W	W	W	W	W		W	W	W	W	W	
	W	W	W	W	W	Up	W	W	W	W	W	3.33%
	W	P	F	X	W		W	P	F	X	W	
	W	B	F	B	W		W	B	F	B	W	
	W	X	F	F	W		W	X	F	F	W	
	W	W	W	W	W		W	W	W	W	W	
	W	W	W	W	W	Left	W	W	W	W	W	3.33%
	W	P	F	X	W		W	P	F	X	W	
	W	B	F	B	W		W	B	F	B	W	

Figura 2. An extract from transition function file. Full function can be found here: https://github.com/mvadrev/init/blob/main/tra_func.pdf

flattenedState = Map.flatten(order='C') flatten row by row

Wall	Wall	Wall	Wall	Wall
Wall	Player	FloorDot	Floor	Wall
Wall	BoxIncorrect	BoxIncorrect	Floor	Wall
Wall	FloorDot	Floor	Floor	Wall
Wall	Wall	Wall	Wall	Wall

4. THE ACTION SET

The actions as A consists of actions Up, Down, Right and Left. We can easily represent this as a discrete set in Python as follows:

A = 0,1,2,3 :: Up, Down. Right, Left = [0,1,2,3]

Since we are dealing with discrete action space and not a continuous one, we also have to make sure we define this in our custom environment class:

self.action_space = spaces.Discrete(6)

At every time step, based on the balance in exploration vs exploitation, a certain action is picked that maximized the overall reward Gt.

5. THE TRANSITION FUNCTION.

The transition function consists of the current state, action and the next state. In our training routine, we have to be able to access this data structure easily. To do this we can represent the table in a csv file and access the particular states, actions and next states. Pandas Python library offers good functions to easily achieve this. Another way to achieve this is to store the transition table as a JSON object. This can help increase the speed of the search as initially we are looking only for the current state in the object. Once the current state is found, we can access the specific action in the JSON subkey and its value as the next state as seen below.

```
[ ]  1 transistion = [{
     2
     3     '0,0,0,0,0,0,5,1,2,0,0,3,1,3,0,0,2,1,1,0,0,0,0,0,0': {
     4         'down': {
     5             'next': '0,0,0,0,0,0,1,1,2,0,0,5,1,3,0,0,4,1,1,0,0,0,0,0,0',
     6             'reward': 5,
     7             'prob': 0.9
     8         },
     9         'right': {
    10             'next': '0,0,0,0,0,0,1,5,2,0,0,3,1,3,0,0,2,1,1,0,0,0,0,0,0',
    11             'reward': -1,
    12             'prob': 0.033
    13         },
    14         'up': {
    15              'next': '0,0,0,0,0,0,5,1,2,0,0,3,1,3,0,0,2,1,1,0,0,0,0,0,0',
    16             'reward': -1,
    17             'prob': 0.033
    18         },
    19         'left': {
    20              'next': '0,0,0,0,0,0,5,1,2,0,0,3,1,3,0,0,2,1,1,0,0,0,0,0,0',
    21             'reward': -1,
    22             'prob': 0.033
    23         }
    24     },
    25
    26     '0,0,0,0,0,0,1,1,2,0,0,5,1,5,0,0,4,1,1,0,0,0,0,0,0': {
    27         'right':  {
```

Figura 3. Transition table JSON object for fast lookup

```
21   def reset(self):
22     self.currentState = np.array(self.map, dtype='int').reshape(1,25)
23     self.done = False
24     return self.currentState
```

Figura 4. Reset method

6. THE ENVIRONMENT: (new)

For the midterm the custom Gym environment was successfully created and the step function and reset functions were coded. For the reset method, we simply take a current copy of the state and return it as a list of integers as seen in Figure(4). For our action space, we used the Gym spaces discrete actions with size of 4. And for our our observation spaced, we used the Gym Box function with a low value of 1 reflecting the lowest value of our states and 5 for the highest, and a shape of (1,25), which is the reason we reshape the lists to 1,25 to keep everything to constant shape.

We also implemented the step function which takes an action and does the following. Figure (5) shows our step function. We also do not update the next state inside the Gym environment but do it in the Q learning algorithm section.

- Calculate the next state and reward by looking up the transition table we formulated as a JSON object.

- Returns the next state, rewards, dones and metadata. The states are stored as a string and converted to an integer array of shape (1,25) whenever needed, to enable searching for a string key in the JSON.

```
def step(self, action):
    ac = self.actionMap[action]
    stringCurrentState = ','.join(map(str, self.currentState[0]))
    next_state = self.transition[0][stringCurrentState][ac]['next']
    list_of_integers = np.array(list(map(int, next_state.split(',')))).reshape(1,25)
    reward = self.transition[0][stringCurrentState][ac]['reward']
    # if(reward == 10):
    #   pass
    print(self.currentState, list_of_integers)
    self.currentState = list_of_integers
    return list_of_integers , reward, self.done, {}
```

Figura 5. step method

In our environment, we have all the states from our transition table in a JSON object. The states are encoded as keys of the JSON object for easy look up. Each state has a subkey holding 4 actions. Each of the 4 actions have the next state reward and probabilities. The gym observation space however expects these variables to be of a certain shape and be integers. So, whenever needed for such functions, we convert the string into integer list. We also converted the transition states to an integer multidimensional array of shape n*1*25. This will also be useful when checking if we are in the terminal state by keeping an integer array of shape 1,25 in the Q-learning algorithm and compared with the state returned by the gym environment's step method.

7. REWARD FUNCTION:

The reward function provides a numeric reward as the agent transitions using the transition function to the next state. In our problem statement we need the agent to quickly move all the boxes to the marked tiles to achieve the subrewards and the terminal reward upon moving the last box correctly. For this a simple reward scheme could be as follows:

- A -1 reward for each step other than when moving boxes onto the marked tiles.
- A +5 reward for moving a box correctly onto a marked tile.
- A +10 reward for moving the last box onto the marked tile.
- A -10 reward for reaching a state where the player is unable to move any boxes because a wall may be blocking it for example.

8. THE ALGORITHM: (NEW):

In our previous report, we stated we will implement PPO as our training algorithm. However, we instead decided to go for Q-learning to complement the theory track happening in class. Q- learning belongs to a class of TD off-policy model free reinforcement learning algorithm methods and hence can solve an MDP if it is given enough time to explore through its episodes. Q learning also being a model free method, does not require the transition probabilities of the environment and actions. Moreover since Q learning is a TD method, unlike in Monte Carlo where the update happens at the end of the episode, in TD learning we can update at every time step thereby guaranteeing convergence given enough time steps. Moreover, it is also relatively easy to implement this algorithm from scratch.

Figura 6. Full reward function can be found here: https://github.com/mvadrev/init/blob/main/rew_func.pdf

In our implementation, we create a class called Q Learning and initialize an empty Q table. We initialized the Q-table with the states in our transition table with 4 actions for each of the states and their respective Q values as zero. We provide the agent with the initial state from our transition table and run the Q learning algorithm on it and let the agent traverse each state until it eventually finds the terminal reward. There is a -1 reward for each time step to encourage the agent to explore. We also know that our map can be solved in less than 10 time steps for our 5x5 grid world as seen below. So, we reset the game every time steps if the terminal reward is not reached by then.

We implemented several methods to help with the Q-learning algorithm. We have a isTerminal state method which returns a boolean operator of True if the agent is in the terminal state. The Q learning algorithm terminates when the agent reaches the terminal state. We also implemented a getNewQ method which takes in oldQ, reward, maxQS-prime to output a newQ value. The maxQ is nothing but the action and its value that brings the maximum return in the next state. In Q learning we use the value of the one step look ahead to calculate the new Q value. Finally we have a learn and predict method.

In the Learn method, we provide the agent with the starting state and traverse through the map and update values until the terminal state has reached. I have not quite finished the predict method but for the most part the logic is complete. Now I just have to check if the values being returned are accurate and the Q values are updating correctly. We use the step method we used in the Gym custom environment we have built to get the next states and rewards to allow the agent to traverse past the initial provided state. We are yet to finish the learning method. The predict method can be seen as follows:

We also have an epsilon greedy action section method that chooses a random action if epsilon is greater than the random probability or chooses the max action for that given state from the Q table as seen below:

9. CONCLUSION

Sokoban has been successfully modeled as an MDP and the agent, state, transition function and reward function were defined. A transition function for the 5x5 grid world was

```
def learn(self, timesteps):
  self.env.reset()
  initialState = self.currentState
  print("Reset success")
  for timestep in range(timesteps):
    while (self.done == False):
      stringCurrentState = ','.join(map(str, initialState[0]))
      action_selection = self.epsilonGreedy(stringCurrentState)
      indexOfS = self.stateTable.index(stringCurrentState)
      print("Index is", indexOfS)
      currentQ = self.qtable[indexOfS]

      for action in self.env.actionMap:
          # print("Action",self.env.actionMap[action])
          next_state , reward, dones,meta = eenv.step(list(eenv.actionMap.values()).index(eenv.actionMap[action]))
          # print("here", next_state , reward, dones)
          stringCurrentState2 = ','.join(map(str, next_state[0]))
          maxQ = self.getMaxQ(stringCurrentState2)
          print("maxQ", maxQ)
          newQ = self.getNewQ(currentQ, maxQ)
          print(newQ)
```

Figura 7. Learn method

```
def getMaxQ(self, sPrime):
  values = []
  for action in self.env.actionMap:
    print("action is", action)
    next_state , reward, dones,meta = self.env.step(action)
    indexOfSPrime = self.stateTable.index(sPrime)
    currentQ = self.qtable[indexOfSPrime]
    values.append(currentQ)
  max_action = np.argmax(values)
  max_action_value = values[max_action]
  return max_action, max_action_value

def epsilonGreedy(self, currentState):
  randomProb = np.random.random()
  stateIndex = self.stateTable.index(currentState)
  if (randomProb > self.epsilon):
    print("exploring...")
    actions = [0,1,2,3]
    randomChoiceAction = np.random.choice(actions)
    return randomChoiceAction
  else:
    print("exploiting.. Choosing best action from Qtable", np.argmax(self.stateTable[stateIndex]))
    return np.argmax(self.stateTable[stateIndex])
```

Figura 8. Other helper methods

```
def isTerminalState(self, currentState):
  if(currentState == self.terminalState):
    return True
  else:
    return False

def getNewQ(self, oldQ, reward, maxQSprime):
  return oldQ + self.alpha * (reward + self.gamma(maxQSprime - oldQ))
```

```
class QLearning():
  def __init__(self, env):
    super(QLearning, self).__init__()
    self.stateTable = [list(eenv.transition[0].items())[i][0] for i in range(len(list(eenv.transition[0].items())))]
    self.qtable = np.zeros([np.array(self.stateTable).shape[0], 4])
    self.epsilon = 0.1
    self.alpha = 0.01
    self.gamma = 0.9
    self.timesteps = 1000
    self.env = env
    self.terminalState = '0,0,0,0,0,0,1,1,2,0,0,1,1,5,0,0,2,1,1,0,0,0,0,0,0'
    self.done = False
```

Figura 9. TerminalState checker method and init variables

modeled keeping in mind that the player can spawn in multiple available spaces on the map. We also implemented the Gym custom class and checked it stableBaseLines3 checkEnv. We have also finished most of the implemetation for our Q-learning algorithm and have to debug and confirm it works without any errors.

What apart remains is making sure everything is running correctly without errors and add some final things to the environment such as reset after 10 steps or reset the environment when terminal reward has reached.

10. CONCLUSION

The final implementation is yet to be complete. But however, to showcase the completeness of the overall project, we have provided a main.py file using which we showcase these various methods we implemented such as sampling random actions 4 times and passing them to the step function, running the rest method returns original map, showing output of epsilon greedy selection on a random state, getMaxQ etc.We hope to finish the whole project by 7th of November.

REFERENCES

1. Sutton Barto. (n.d.). Introduction To Reinforcement Learning. Retrieved October 25, 2022, from http://incompleteideas.net/book/RLbook2020.pdf