CS 271

Sokoban Reinforcement Learning

By Group 19

Jonathan Dedinata, 64003232

Ting-Ta Chung, 22186924

Zhecheng Huang, 60402375

Contents

I. Problem

For this project, we are trying to solve the Sokoban puzzle using Artificial Intelligence. A Sokoban puzzle board consists of a player, a number of boxes, a number of target spaces where the boxes are supposed to be pushed to, and walls. The goal of the puzzle is to push each of the boxes into target spaces located around the board.

Our goal is to make our agent solve the Sokoban puzzle by itself using Reinforcement Learning. With reinforcement learning, our agent should be able to learn from the process, improve from the errors, and finally converge to find the optimal solution.

II. Our Approach

Our approach to the problem is to use Q-Learning as a way for the agent to learn the most optimal path towards a solution. We used the Q-Table as our state space and updated the Q-Table dynamically throughout the iteration of finding a solution for a Sokoban board.

At the beginning of an iteration, the Q-Table is empty and will only be updated when the agent makes an action. We used the Manhattan Distance function to determine the reward value of a certain action and used a greedy heuristic such that the agent will choose to move towards the most rewarded direction. Should the agent be presented with choices that have the same reward value, the direction will be chosen at random. Additionally, the agent will have a possibility to move into an unintended direction in order to prevent the agent from making repeated choices.

We think that our approach will work as the agent will move towards the most efficient direction. Since the Q-Table is initialized to be all 0, the agent will start by making random moves. However, everytime the agent pushes the box into the correct path, the agent will be rewarded with positive values and in the end, the choices should converge and find a solution for the Sokoban board.

III. Algorithm Design

In our algorithm, we implemented Q-Learning with a greedy heuristic. Within a state space, we evaluate actions that the agent has taken and update the Q-Table accordingly. When the agent pushes a box into a target space, the agent will be rewarded. However, for every action the agent makes, the agent will be penalized by the number of total steps the agent has taken. Additionally, when the agent makes a mistake, such as walking into a wall or pushing a box into a corner, the agent will be further penalized.

During the initialization of the Q-Table, it is initialized as an empty 2-Dimensional array. On step 1 of the reinforcement learning iteration, the current state is pushed into the Q-Table, with every reward value initialized as 0. The agent will then take an action, updating the Q-Table by adding the reward value of the action that the agent took by using the Manhattan Distance. Our formula for Manhattan Distance is:

$$Manhattan\ Dist(b_i, p_j) = |b_{ix} - p_{jx}| + |b_{iy} - p_{iy}|$$

and our formula for calculating the reward value of an action is:

$$R = -\sum_{i=0}^{n} min \sum_{j=0}^{n} Manhattan\ Dist(b_i, p_i)$$

The agent will then check the Q-Table for the highest reward value in the current state and choose to move in that direction. However, to avoid the agent from making repeating moves without progress, we added a randomness element. We set a hyperparameter, **ε**, to decide whether the agent can take the action with the highest reward value. In our project, we tested with **ε** = 0.85, which means that the agent will take the intended direction 85% of the time. However, 15% of the time, it will choose its direction randomly, including the intended direction. Mathematically, the agent will heavily favor moving into the intended direction and this is designed so that the impact of **ε** is minimal since **ε** is only used to prevent an agent from being stuck in a loop.

The agent will then keep looking for a solution until either all the boxes are inside the target space or a box is stuck in a corner while not inside the target space. For every action the agent

makes, the reward value is also updated so that the value is negative the total number of steps that the agent has taken.

$$R \; -= \; totalSteps$$

This is to incentivize the agent to not repeat unsuccessful actions that have been taken before. Additionally, we also subtract the reward value by a large number if the agent takes an unnecessary action such as walking into a wall or pushing a box that has been pushed into a target space. The reward value will then be further processed with other hyperparameters to evaluate the value to be pushed into the Q-Table. The formula that we used is as follows:

$$nextQ \; = \; R \; + \; \gamma \; \times \; max(qValue_{k+1})$$

$$curQ \; = \; qValue_{k}$$

$$qValue_{k} \; += \; \alpha \; \times \; (nextQ \; - \; curQ)$$

In our formula, α and γ are hyperparameters that we change. α represents the learning rate of the agent and γ represents the discount factor of future actions. $qValue_{k}$ represents the values of the Q-Table in the current state and $qValue_{k+1}$ represents the values of the Q-Table in the next state.

IV. Description and Analysis of Properties

Our Q-Table is initialized using a Panda DataFrame, so each search of the Q-Table is of time complexity O(1). Updating the Q-Table is also O(1) since it behaves like a Python Dictionary. Since the Q-Table is updated dynamically, new entries to the Q-Table are created only when the agent makes the action, resulting in less space being used.

We expect the space complexity of the Q-Table to be O(S * A) with S being the number of states and A being the number of actions. S represents the positions of the boxes. If we assume the playboard length is L. The positions of the boxes will be O(L^2) which means O(S) = O(L^2). On the other hand, there are only 9 actions in total. In conclusion, O(S * A) = O($9L^2$) = O(L^2). The space complexity is O(L^2) with L representing the length of the playboard.

Estimating the number of steps the agent will take will be difficult due to the random nature of our algorithm. In the worst case scenario, the agent will take every possible action without

solving the Sokoban board even once, which is O(A*E) where A is the largest number of steps the agent can take in a single episode and E is the number of episodes run in the simulation to try and get a solution. The Q-Table will then be extremely large. However, since the agent will try to find a solution, it is not feasible to base our complexity on the worst case scenario as the agent will keep learning and certain actions that lead to negative outcomes will be pruned naturally.

However, for the combination of time complexity of our functions, we expect the total time complexity to be $O(Nm^2)$ with N being the number of times we decide to iterate to train the agent and m being the number of boxes in the environment.

The values of the hyperparameters, α, γ, and ε, in the project are obtained through trial-and-error. We concluded that in the benchmark sokoban01.txt, we can use the values $\alpha = 0.1$, $\gamma = 0.9$, and $\varepsilon = 0.85$ so the agent can solve the Sokoban puzzle in a reasonable amount of time.

V. Performance Assessment

Through our experiment results and observations, we found that there might be some problems within our original rewarding system. Our original rewarding system is as follows:

$$Manhattan\ Dist(b_i, p_j) = |b_{ix} - p_{jx}| + |b_{iy} - p_{iy}|$$

$$R = -\sum_{i=0}^{n} min \sum_{j=0}^{n} Manhattan\ Dist(b_i, p_i)$$

According to our experience, this system has higher probability to help us find our solution quickly but it is unstable. And also, it is hard to converge to the optimal solution in the end.
We could reasonably infer that we give our agent a penalty in each step because the Manhattan Distance of it is greater than zero. Therefore, if our agent goes into a new iteration after successfully solving the board, it initializes the new state in Q-Table with zero in the directions that the agent has never taken before. Since the value corresponding to the old state and action in Q-Table is negative, and the value corresponding to the new state and action is zero, the agent will then keep trying to find a new solution instead of going back to the previous found solution.

Therefore, we slightly modified it and built our second version, it is as follows:

$$curDis = -\sum_{i=0}^{n} min \sum_{j=0}^{n} Manhattan\ Dist(b_i, p_i)$$

$$R \mathrel{+}= (preDis - curDis)$$

$$Manhattan\ Dist(b_i, p_j) = |b_{ix} - p_{jx}| + |b_{iy} - p_{iy}|$$

Based on our observation, this system could prevent our agent from finding a new route, however, it is easy to get stuck in the local minimum.

Finally, after multiple experiments and full consideration, we decided to not use Manhattan Distance in our rewarding system. Although it took a longer time to find the first solution, it is stable and able to converge in the end.

However, in benchmarks with a large board size and a large number of boxes, our algorithm fails to solve the problem within a reasonable amount of time. We expect this is because the reward system is not set up in a way that it can properly detect when an agent is stuck. It also currently does not reward the agent with a positive value when the agent pushes the box in the right direction. This results in the agent making random moves until it pushes one box into a target space. Running the algorithm against benchmarks besides sokoban00.txt and sokoban01.txt results in a very long runtime.

In the case of sokoban00.txt and sokoban01.txt, our algorithm works as the boxes are not lined up against each other. Each box is also very near to target spaces, with each target space also far from each other. This results in a clearer path to the goal. However, in other benchmarks such as sokoban02.txt and sokoban03.txt, the boxes are lined up against each other and our algorithm currently cannot efficiently detect whether a box has been pushed to a state where it can no longer be pushed, thereby needing to reset early. This results in the agent making moves even after it is stuck and can no longer solve the board in an episode. In conclusion, we expect our algorithm to excel under conditions where the boxes are not lined up against each other and where the target spaces are spaced out between each other.

VI. Our Observations

Our observations are as follows:

Benchmark: sokoban01.txt

When Q-Table is preserved:

Iteration	Time	Space Used	γ	α	ε	Episodes
1	19.50	1256	0.9	0.1	0.85	18
2	3.12	1428	0.9	0.1	0.85	2
3	7.71	1672	0.9	0.1	0.85	7
4	22.8	2072	0.9	0.1	0.85	10
5	4.32	2256	0.9	0.1	0.85	3
6	2.04	2284	0.9	0.1	0.85	2
7	0.39	2284	0.9	0.1	0.85	2
8	8.12	2396	0.9	0.1	0.85	3
9	3.06	2412	0.9	0.1	0.85	2
10	1.13	2412	0.9	0.1	0.85	1
11	4.62	2628	0.9	0.1	0.85	2
12	6.6	2716	0.9	0.1	0.85	2
13	2.33	2716	0.9	0.1	0.85	1
14	0.55	2716	0.9	0.1	0.85	1
15	3.81	2716	0.9	0.1	0.85	1

When Q-Table is not preserved:

Iteration	Time	Space Used	γ	α	ε	Episodes
1	8.03	708	0.9	0.1	0.9	7
2	14.25	1216	0.9	0.1	0.85	10
3	28.99	1920	0.9	0.1	0.8	28

Iteration	Time	Space Used	γ	α	ε	Episodes
1	18.8	1452	0.95	0.1	0.9	19
2	8.03	708	0.9	0.1	0.9	7
3	28.99	2072	0.85	0.1	0.9	31

Iteration	Time	Space Used	γ	α	ε	Episodes
1	6.93	828	0.9	0.2	0.9	11
2	9.54	764	0.9	0.15	0.9	8
3	8.03	708	0.9	0.1	0.9	7

From the observations and results above, we can see that the agent can solve sokoban01.txt in a reasonable amount of time, even after finding a solution. In our original algorithm, when we keep using the same Q-Table without re-initializing, the episodes and time increase as the iteration keeps going. However, after modifying the algorithm, we obtained the results above. Time is measured in seconds and Space Used is the length of the Q-Table at the end of the iteration after the agent has successfully found a solution. Episodes are the number of resets that the agent has to do before finding a solution.

VII. Appendix

Example of a Q-Table:

	a1	a2	a3
s1	r11	r12	r13
s2	r21	r22	r23

s_i : the position of the boxes

a_i : move up, move down, move left, move right

r_i : the reward that the agent will get in a specific state and action

Summary of what the Q-Learning function does:

Q_Learning():

1. Choose action based on the current state
2. Agent takes action and get the rewards and the next state
3. Agent uses Q-learning to learn from this experience
4. If it finished, agent or box get stuck in the corner, break
5. Do the following steps for n times

Pseudocode of Algorithm

Sokoban():

Training Agent Section
Initialize Q(s, a)
for episodes in range(N):
  totalSteps = 0
  state = env.reset()
  while True:
    action = chooseAction(state)
    totalSteps++
    next_state, reward, finished = evaluateAction(action, totalSteps)
    Q_learning(state, action, reward, next_state)
    state = next_state
    # if it finished, agent or box get stuck in the corner, break
    if finished or stuck in corner:
      break

chooseAction(state):
  pick a random possibility from 0 to 1
  if(possibility < p):
    foreach action a_i:
      action = argmax(Q(state, a_i))
  else:
    action = randomly choose from the actions set
  return action

Manhattan_Dis(b_i , p_j):
  return $\left|b_{ix} - p_{jx}\right| + \left|b_{iy} - p_{jy}\right|$

evaluateAction(action, totalSteps):
  Move player according to the action
  next_state = the current position of the boxes
  finished = false
  for i = 1 to m: # iterate over the number of boxes
    Set r to infinity
    for j = 1 to m: # iterate over the number of goals
      r = min(r, Manhattan_Dis(box_i , $position_j$))

```
        reward = reward - r
    if(reward = 0):
        finished = true
    reward = reward - totalSteps
    return next_state, reward, finished

Q_learning(state, action, reward, next_state):
    Set nextQ to -infinity
    if next_state is not terminal:
        for each action a_i:
            nextQ = max(nextQ, Q(next_state, a_i)
        nextQ = reward + γ * nextQ
    else nextQ = reward
    Q(state, action) = Q(state, action) + α*(nextQ -  Q(state, action))
```

Source Code Github Link:

https://github.com/Jonathan-Dedinata/CS271Sokoban/tree/final

VIII. References

1. https://www.learndatasci.com/tutorials/reinforcement-q-learning-scratch-python-openai-gym/
2. https://github.com/mpSchrader/gym-sokoban
3. https://mofanpy.com/tutorials/machine-learning/reinforcement-learning/