CS271 Project: Sokoban Solver using Q-Learning and BFS

December 2021

Contents

1 Team: **2**

2 Overview **2**

3 Programming Language **2**

4 Libraries **2**

5 Environment Setup **2**

6 Q-Learning **4**

7 Q-Learning Algorithm **5**

8 Initialisation of Q Values **5**

9 Find whether there is path between two cells **5**
- 9.1 Simple and Efficient solution 5
- 9.2 Approach 6
- 9.3 Algorithm 6

10 Experiments and Result Analysis **6**
- 10.1 Complexity of Sokoban problem 6
- 10.2 Why our Q-learning approach works 7
- 10.3 Experimentation 7

11 Benchmark test case statistics **9**

12 Limitations **11**
- 12.1 Why our algorithm doesn't work for hard challenges? 11

13 Conclusion **12**

14 Code link **12**

A Appendix **12**
A.1 Pseudocode used for our implementation of Sokoban: 12
A.2 Custom Input Solutions 17

1 Team:

Name	Email	Student ID
Hari Kishore Chaparala	hchapara@uci.edu	48481412
Vishakha Prashant Ratnaparkhi	vratnapa@uci.edu	78685485
Rutuja Tushar Pansare	rpansare@uci.edu	55443146

2 Overview

We aim to implement the Q-learning algorithm to build our Sokoban Solver. We define the states, termination conditions, actions and rewards at first. We will then reduce the state space by a factor of N*M*L (board length x board width x average number of moves the player can make) using BFS. Additionally, we will randomly select amongst the best actions and a random action using ϵ-greedy approach. We are only trying to find a path and not the optimal path (solution with least number of moves) here. Further, we describe our proposed algorithm, a brief explanation of the algorithm and steps we took to implement it, followed by various experiments with the heuristic parameters and conclusions drawn. We are also able to solve 6/96 benchmark test cases with our approach under 1 hour.

3 Programming Language

Python

4 Libraries

No external libraries are used.

5 Environment Setup

In this section, we define the state, termination conditions, permitted actions, and rewards. We also specify the initialisation of the Q-values.

Definitions
N: board rows
M: board columns
P: Player
Q: Q value
S: State
S': next state
A: action
R: reward

α: learning rate
γ: discount factor
B: Boxes
G: Storage locations
W: walls

- **State(S):** We only use the box's location as state. The idea is that if a player is at position (x1, y1) and we want to see if the player can move a box at location (x2, y2), we simply have to find if there is a path without any blockers connecting the player location to any adjacent location of the box. Note that we have added capability to turn off the BFS, thus also adding the player location to the state.
- **Action(A):**

  If we define action as each movement by the person/player (up, down, left, right), this would cause our state space to become unnecessarily large. We need our solver to move the boxes to storage locations. Therefore, to reduce the size of the Q-Learning table, we define an action as moving a box in a direction it is allowed to move in. A box is said to be allowed to move when there is no obstruction in the desired direction. An obstruction could be a wall or another box itself. Note that we have added capability to turn off the BFS, thus by also making the player movement as an action.
- **Deadlock:** Deadlocks are game states that can never lead to a goal state. Deadlocks are extremely important in Sokoban. They are very common and greatly impact the performance of the solution. We are at a deadlock if there are no possible actions or if a box is stuck in a non-storage location. We implemented detection of two types of deadlocks
  1. **Dead Square Deadlock** Dead squares are spaces that are always a deadlock if there is a box at that location. For example, if you push a box into a wall that doesn't have a storage location, there is no way to push the box off the wall to get it to a storage location.
  2. **Freeze Deadlock** A freeze deadlock is when a box becomes immovable and is not a storage location. A common example is pushing a box into a corner, but a box can also be frozen without any walls involved.
- **Termination condition:** To form a favourable termination condition, we need to place all boxes in one of the storage locations. When a box (any box) is not in a storage location but cannot be moved is called unfavourable termination condition. By assigning appropriate rewards for these termination/goal states, we propose to differentiate between these two types : a favourable reward for placing a box in a storage location, and a reward to represent negative infinity for ending up with a box that cannot be moved.
- **Reward(R):** Rewards are defined in order to decide between the probable actions. A more desirable action will have a higher reward and a less one will have a higher negative reward. For instance, we can have a reward for each movement of the player, a higher reward for the player moving the box and a yet higher reward if the box is moved to the storage location.
  In our proposed solution, as our actions are defined by moving a box, we don't define reward for player movement (Using BFS, we can see if the player can be moved to a location adjacent to the box). If BFS is turned off, player movement is also given reward of ± 1 depending on the closeness to any box.

Following are the scenarios. The reward values are just for reference.

1. Storage to Storage: -10
2. Storage to non-Storage: -100
3. Non-storage to non-storage:
    (a) Moving closer to the nearest storage location: +10
    (b) Moving farther from the nearest storage location: -10
4. Non-storage to storage: +100
5. Box deadlocked and not in storage location: (We directly assign Q value for this. See below in Q -Learning section)
6. All boxes in storage locations: (We directly assign Q value for this. See below in Q-Learning section)

6 Q-Learning

Q-learning is an off policy reinforcement learning algorithm that seeks to find the best action to take given the current state. Q-learning seeks to learn a policy that maximizes the total reward. It's considered off-policy because the q-learning function learns from actions that are outside the current policy, like taking random actions, and therefore a policy isn't needed. When q-learning is performed we create what's called a q-table or matrix that follows the shape of [state, action] and we initialize our values to zero. We then update and store our q-values after an episode. This q-table becomes a reference table for our agent to select the best action based on the q-value.

Updating q values:

Agent takes an action using 2 ways:

1. **Exploiting:** Using information available to us in order to make a decision. Agent uses the q-table as a reference, views all possible actions for a given state and then selects the action based on the max value of those actions.
2. **Exploring:** Acting randomly. It allows the agent to explore and discover new states that otherwise may not be selected during the exploitation process.

ϵ greedy approach can be used to balance between these two to set the value of how often you want to explore vs exploit.

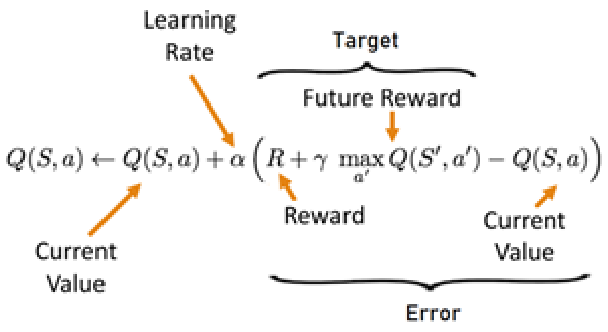
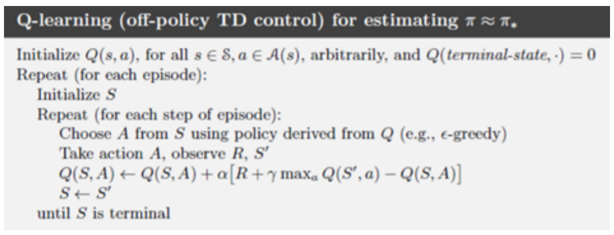

Learning Rate
Target
Future Reward
$Q(S,a) \leftarrow Q(S,a) + \alpha \left(R + \gamma \max_{a'} Q(S',a') - Q(S,a)\right)$
Reward
Current Value
Current Value
Error

Figure 1: Q Learning Equation

7 Q-Learning Algorithm



```
Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(terminal\text{-}state, \cdot) = 0$
Repeat (for each episode):
   Initialize $S$
   Repeat (for each step of episode):
      Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
      Take action $A$, observe $R$, $S'$
      $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$
      $S \leftarrow S'$
   until $S$ is terminal
```

Figure 2: Q Learning Algorithm

8 Initialisation of Q Values

Terminal state Q value: MAX INT value
Deadlock state: MIN INT value

9 Find whether there is path between two cells

BFS Breadth first search is a graph traversal algorithm that starts traversing the graph from root node and explores all the neighbouring nodes. Then, it selects the nearest node and explore all the unexplored nodes. The algorithm follows the same process for each of the nearest node until it finds the goal.

9.1 Simple and Efficient solution

The graph is the matrix in itself.

9.2 Approach

The idea is to use Breadth-First Search on the matrix itself.Consider a cell=(i,j) as a vertex v in the BFS queue. A new vertex u is placed in the BFS queue if u=(i+1,j) or u=(i-1,j) or u=(i,j+1) or u=(i,j-1) and it is an empty cell. Starting the BFS algorithm from cell=(i,j) such that M[i][j] is the player location and stopping either if there was a reachable vertex u=(i,j) such that M[i][j] is the destination cell and returning true or every cell was covered and there was no such a cell and returning false.

9.3 Algorithm

1. Create BFS queue q with player location as the element.
2. Run BFS algorithm with q, skipping cells which are not valid. i.e: they are walls or boxes or outside the matrix bounds and marking them as visited upon successful visitation.
    (a) If in the BFS algorithm process there was a vertex x=(i,j) such that M[i][j] is destination cell stop and return true.
    (b) BFS algorithm terminated without returning true then there was no element M[i][j] which is destination, then return false.

10 Experiments and Result Analysis

10.1 Complexity of Sokoban problem

Sokoban is a challenging one-player puzzle game in which the goal is to navigate a gridworld maze and push box-like objects onto goal tiles. A Sokoban puzzle is considered solved when all boxes are positioned on top of goals. The difficulty of Sokoban puzzles emerges from its constraints: the player can move in all 4 cardinal directions, but cannot pass through the walls of the maze or boxes. The player can push adjacent boxes when there is an empty space behind the box, but cannot pull boxes.

Despite its simple ruleset, Sokoban is an incredibly complex game for which no general solver exists. Sokoban is $NP-Hard$ and $PSPACE-complete$ and has an enormous state space that makes it inassailable to exhaustive search methods. An efficient automated solver for Sokoban must have strong heuristics so that it is not overwhelmed by the number of possible game states. Another reason why Sokoban is difficult is that moves in the game are sometimes non reversible, which may lead to situations where the solution is not reachable from the current game state. When this occurs, the game is said to be in a state of deadlock. This property distinguishes Sokoban from other puzzles such as sliding puzzles or Rubik's Cube, where moves are always reversible. In the general case, detecting whether a Sokoban puzzle is deadlocked is as hard as solving the puzzle, which makes deadlock detection NP-Hard as well.

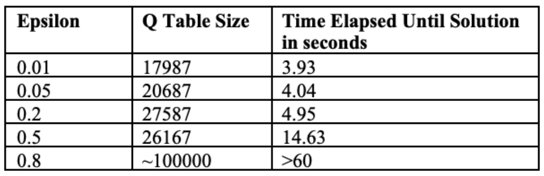
10.2 Why our Q-learning approach works

Reinforcement learning is a generalization of the supervised learning paradigm, in which the underlying distribution of the data shown to the decision-making agent is non-stationary, and some aspects of the data may be occluded from the perspective of the agent. Model-free reinforcement learning methods are capable of achieving superhuman performance in environments where precise control and high reaction speeds are of paramount importance. However, in environments where long-term planning and careful deliberation are important, it becomes necessary to augment reinforcement learning with search-based methods such as Monte-Carlo Tree Search. In our ϵ-greedy based Q-learning approach, our heuristics are able to construct proper Q-values for boards up to size 20x20 (excluding the tricky/challenging test cases) which help in finding a quick solution. However, because of the nature of heuristics, there can be some wasted moves with a player stuck at a local maxima. Due to some non-deterministic nature of our algorithm, the player can come out of some of those loops and cyclic movements.

10.3 Experimentation

We conducted experiments by varying various heuristic parameter values and the observations as below:

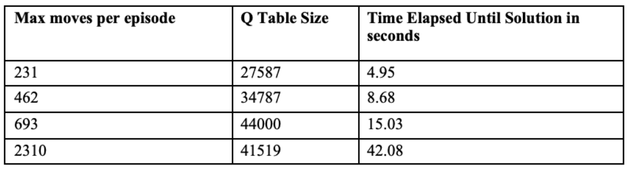
1. **By varying Epsilon** With higher epsilon value, the Q table size tends to be higher, which results in more time for algorithm to find a solution. A higher epsilon value means the model will explore more which will tend to increase the size of table. Also, we cannot depend on lower values of epsilon even though the time required is less because it can have a trade off for tough boards. Therefore, we consider 0.2 as optimal value of epsilon as very low value will not work for tricky cases.



Epsilon	Q Table Size	Time Elapsed Until Solution in seconds
0.01	17987	3.93
0.05	20687	4.04
0.2	27587	4.95
0.5	26167	14.63
0.8	~100000	>60

Figure 3: Variation in Epsilon

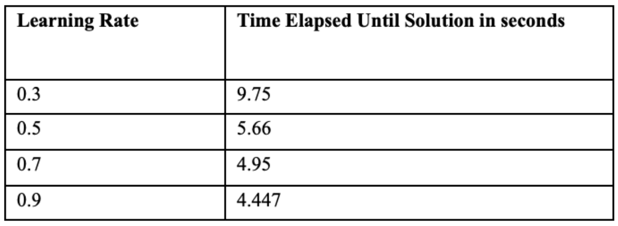
2. **By varying Maximum Step Size** The max moves made in each episode will be decided by box dimensions and number of storage locations. When the max moves are high, the algorithm tends to explore more, and will cause more Q table size. This leads to divergence i.e Q values will not be good in a local area. The overall effect is that we will consume a lot of time. This case may not be true for tricky boards (where number of moves are limited and exploration is the key)



Max moves per episode	Q Table Size	Time Elapsed Until Solution in seconds
231	27587	4.95
462	34787	8.68
693	44000	15.03
2310	41519	42.08

Figure 4: Variation in Maximum Moves

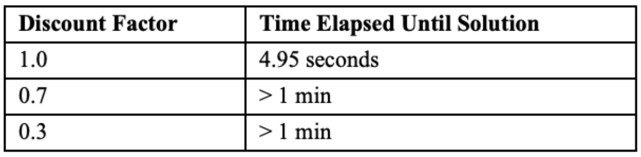
3. **By varying Learning Rate** We conducted the experiment by varying the learning rate and noted the below observations. The algorithm performs better when the learning rate increases. We have observed that the optimal value of learning rate in 0.7.



Learning Rate	Time Elapsed Until Solution in seconds
0.3	9.75
0.5	5.66
0.7	4.95
0.9	4.447

Figure 5: Variation in Learning Rate

4. **By varying Discount Factor** We have observed multiple scenarios by varying discount factor we have concluded that 1.0 discount factor provides with optimal time.



Discount Factor	Time Elapsed Until Solution
1.0	4.95 seconds
0.7	> 1 min
0.3	> 1 min

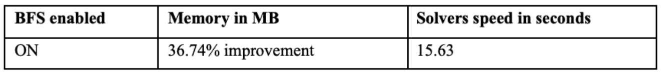
Figure 6: Variation in Discount Factor

5. **Variation in the Reward Values-** Default Reward Values are -1 for each move; ±3 for boxes closer to storage location ; ±15 for boxes in storage, and ±1 for player close to the box.

   We have identified a lot of variation in algorithm performance based on heuristic rewards values. Typically, if absolute rewards values for different scenarios are low, then algorithm tends to come up with better Q values.

   Also, giving heavier rewards to a box when moved to storage location has a negative effect on the time taken to solve Sokoban. This could be because in order to place a box in a storage location, some boxes may be required to be moved from their storage location temporarily.

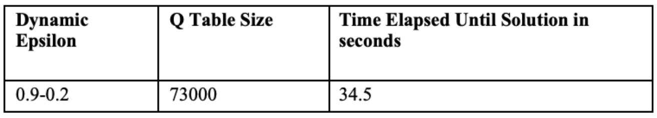
6. **With BFS**
   Using BFS, since we are moving player out of picture, we save a lot of moves and memory. BFS takes n*m time for each episode. But with BFS enabled algorithm performs poorly in terms of time. Also BFS is better suited for tricky Sokoban boards, where there are lot of obstacles .



BFS enabled	Memory in MB	Solvers speed in seconds
ON	36.74% improvement	15.63

Figure 7: Variation with BFS

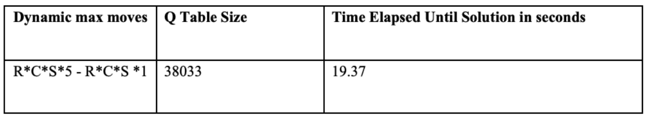
7. **Variation with Dynamic Epsilon**
   In Dynamic Epsilon flow we initiate epsilon with 0.9 and then for every 1000 episodes done, we decrease by factor of 0.9 until 0.2. This approach doesn't make any difference for small board where for bigger boards it can make a significant difference.



Dynamic Epsilon	Q Table Size	Time Elapsed Until Solution in seconds
0.9-0.2	73000	34.5

Figure 8: Variation with Dynamic Epsilon

8. **Variation with Dynamic Maximum Moves**
   In Dynamic max moves flow we initiate max moves with row size * column size * no of storage locations * 5 and then for every 1000 episodes done, we decrease by factor of 0.9 until row size * column size * no of storage locations. This approach doesn't make any difference for small board whereas for bigger boards it can make significant difference.

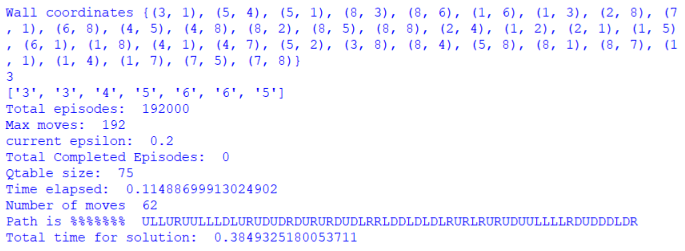


Dynamic max moves	Q Table Size	Time Elapsed Until Solution in seconds
R*C*S*5 - R*C*S *1	38033	19.37

Figure 9: Variation with Dynamic Max Moves

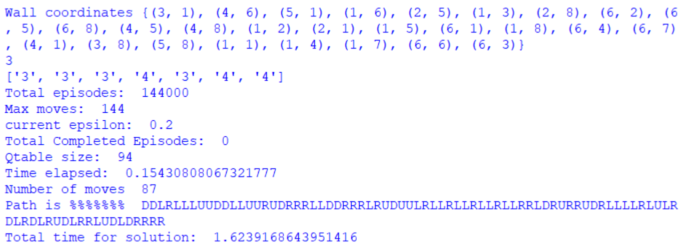
11 Benchmark test case statistics

We are able to solve 6 out of 96 benchmark test cases within the time limit of 1 hour. Below are the results.



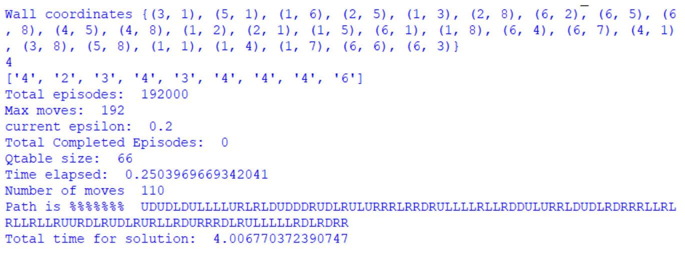
```
Wall coordinates {(3, 1), (5, 4), (5, 1), (8, 3), (8, 6), (1, 6), (1, 3), (2, 8), (7
, 1), (6, 8), (4, 5), (4, 8), (8, 2), (8, 5), (8, 8), (2, 4), (1, 2), (2, 1), (1, 5)
, (6, 1), (1, 8), (4, 1), (4, 7), (5, 2), (3, 8), (8, 4), (5, 8), (8, 1), (8, 7), (1
, 1), (1, 4), (1, 7), (7, 5), (7, 8)}
3
['3', '3', '4', '5', '6', '6', '5']
Total episodes:  192000
Max moves:  192
current epsilon:  0.2
Total Completed Episodes:  0
Qtable size:  75
Time elapsed:  0.11488699913024902
Number of moves  62
Path is %%%%%%%  ULLURUULLLDLURUDUDRDURURDUDLRRLDDLDLDLRURLRURUDUULLLLRDUDDDLDR
Total time for solution:  0.3849325180053711
```

Figure 10: Benchmark input: sokoban-01



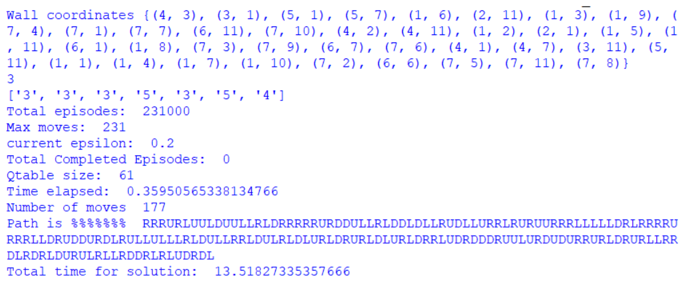
```
Wall coordinates {(3, 1), (4, 6), (5, 1), (1, 6), (2, 5), (1, 3), (2, 8), (6, 2), (6
, 5), (6, 8), (4, 5), (4, 8), (1, 2), (2, 1), (1, 5), (6, 1), (1, 8), (6, 4), (6, 7)
, (4, 1), (3, 8), (5, 8), (1, 1), (1, 4), (1, 7), (6, 6), (6, 3)}
3
['3', '3', '3', '4', '3', '4', '4']
Total episodes:  144000
Max moves:  144
current epsilon:  0.2
Total Completed Episodes:  0
Qtable size:  94
Time elapsed:  0.15430808067321777
Number of moves  87
Path is %%%%%%%  DDLRLLLUUDDLLUURUDRRRLLDDRRRLRUDUULRLLRLLRLLRLLRRLDRURRUDRLLLLRLULR
DLRDLRUDLRRLUDLDRRRR
Total time for solution:  1.6239168643951416
```

Figure 11: Benchmark input: sokoban-02



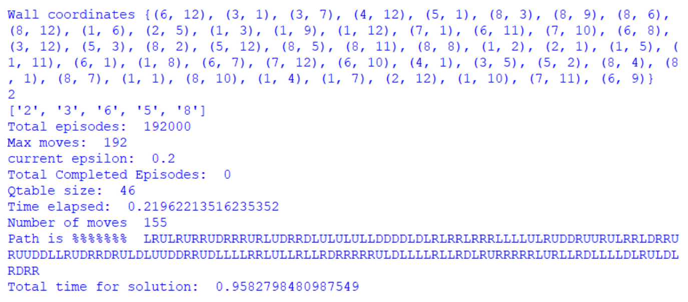
```
Wall coordinates {(3, 1), (5, 1), (1, 6), (2, 5), (1, 3), (2, 8), (6, 2), (6, 5), (6
, 8), (4, 5), (4, 8), (1, 2), (2, 1), (1, 5), (6, 1), (1, 8), (6, 4), (6, 7), (4, 1)
, (3, 8), (5, 8), (1, 1), (1, 4), (1, 7), (6, 6), (6, 3)}
4
['4', '2', '3', '4', '3', '4', '4', '4', '6']
Total episodes:  192000
Max moves:  192
current epsilon:  0.2
Total Completed Episodes:  0
Qtable size:  66
Time elapsed:  0.2503969669342041
Number of moves  110
Path is %%%%%%%  UDUDLDULLLLURLRLDUDDDRUDLRULURRRLRRDRULLLLRLLRDDULURRLDUDLRDRRRLLRL
RLLRLLRUURDLRUDLRURLLRDURRRDLRULLLLLRDLRDRR
Total time for solution:  4.006770372390747
```

Figure 12: Benchmark input: sokoban-03



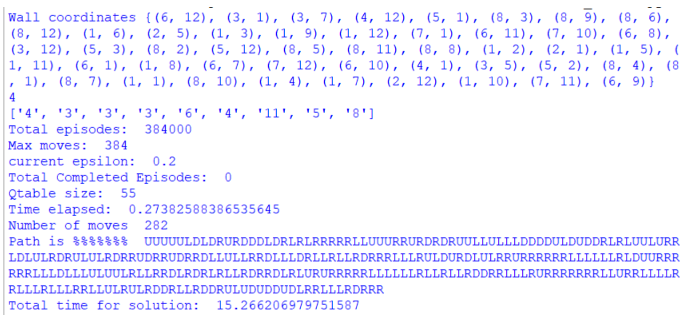
```
Wall coordinates {(4, 3), (3, 1), (5, 1), (5, 7), (1, 6), (2, 11), (1, 3), (1, 9), (
7, 4), (7, 1), (7, 7), (6, 11), (7, 10), (4, 2), (4, 11), (1, 2), (2, 1), (1, 5), (1
, 11), (6, 1), (1, 8), (7, 3), (7, 9), (6, 7), (7, 6), (4, 1), (4, 7), (3, 11), (5,
11), (1, 1), (1, 4), (1, 7), (1, 10), (7, 2), (6, 6), (7, 5), (7, 11), (7, 8)}
3
['3', '3', '3', '5', '3', '5', '4']
Total episodes:  231000
Max moves:  231
current epsilon:  0.2
Total Completed Episodes:  0
Qtable size:  61
Time elapsed:  0.35950565338134766
Number of moves  177
Path is %%%%%%%  RRRURLUULDUULLRLDRRRRRURDDULLRLDDLDLLRUDLLURRLRURUURRRLLLLLDRLRRRRU
RRRLLDRUDDURDLRULLULLLRLDULLRRLDULRLDLURLDRURLDLURLDRRLUDRDDDRUULURDUDURRURLDRURLLRR
DLRDRLDURULRLLRDDRLRLUDRDL
Total time for solution:  13.51827335357666
```

Figure 13: Benchmark input: sokoban-04



```
Wall coordinates {(6, 12), (3, 1), (3, 7), (4, 12), (5, 1), (8, 3), (8, 9), (8, 6),
(8, 12), (1, 6), (2, 5), (1, 3), (1, 9), (1, 12), (7, 1), (6, 11), (7, 10), (6, 8),
(3, 12), (5, 3), (8, 2), (5, 12), (8, 5), (8, 11), (8, 8), (1, 2), (2, 1), (1, 5), (
1, 11), (6, 1), (1, 8), (6, 7), (7, 12), (6, 10), (4, 1), (3, 5), (5, 2), (8, 4), (8
, 1), (8, 7), (1, 1), (8, 10), (1, 4), (1, 7), (2, 12), (1, 10), (7, 11), (6, 9)}
2
['2', '3', '6', '5', '8']
Total episodes:  192000
Max moves:  192
current epsilon:  0.2
Total Completed Episodes:  0
Qtable size:  46
Time elapsed:  0.21962213516235352
Number of moves  155
Path is %%%%%%%  LRULRURRUDRRRURLUDRRDLULULULLDDDDLDLRLRRLRRRLLLLULRUDDRUURULRRLDRRU
RUUDDLLRUDRRDRULDLUUDDRRUDLLLLRRLULLRLLRDRRRRRULDLLLLRLLRDLRURRRRRLURLLRDLLLLDLRULDL
RDRR
Total time for solution:  0.9582798480987549
```

Figure 14: Benchmark input: sokoban-05a



```
Wall coordinates {(6, 12), (3, 1), (3, 7), (4, 12), (5, 1), (8, 3), (8, 9), (8, 6),
(8, 12), (1, 6), (2, 5), (1, 3), (1, 9), (1, 12), (7, 1), (6, 11), (7, 10), (6, 8),
(3, 12), (5, 3), (8, 2), (5, 12), (8, 5), (8, 11), (8, 8), (1, 2), (2, 1), (1, 5), (
1, 11), (6, 1), (1, 8), (6, 7), (7, 12), (6, 10), (4, 1), (3, 5), (5, 2), (8, 4), (8
, 1), (8, 7), (1, 1), (8, 10), (1, 4), (1, 7), (2, 12), (1, 10), (7, 11), (6, 9)}
4
['4', '3', '3', '3', '6', '4', '11', '5', '8']
Total episodes:  384000
Max moves:  384
current epsilon:  0.2
Total Completed Episodes:  0
Qtable size:  55
Time elapsed:  0.27382588386535645
Number of moves  282
Path is %%%%%%%  UUUUULDLDRURDDDLDRLRLRRRRRLLUUURRURDRDRUULLULLLDDDDULDUDDRLRLUULURR
LDLULRDRULULRDRRUDRRUDRRDLLULLRRDLLLDRLLRLLRDRRRLLLRULDURDLULRRURRRRRRLLLLLLRLDUURRR
RRRLLLDLLLULUULRLLRRDLRDRLRLLRDRRDLRLURURRRRRLLLLLLRLLRLLRDDRRLLLRURRRRRRRLLURRLLLLR
RLLLRLLLRRLLULRULRDDRLLRDDRULUDUDDUDLRRLLLRDRRR
Total time for solution:  15.266206979751587
```

Figure 15: Benchmark input: sokoban-05b

12 Limitations

1. Our Q values are initialized to zero. So the Q-learning algorithm takes a while to get to meaningful values for states.
2. Our algorithm tends to get stuck in local maxima. For example -When there are limited moves, player tends to move back and forth or in cycles. We tried to mitigate it by memorizing moves, but there are too many combinations to check for.
3. Our heuristics are pretty much straight forward. As seen in the above results, reward values variation significantly affects the performance. To mitigate this, we need more heuristics so that total reward is distributed and we also need heuristics that are more suited to the Sokoban rather than simple Manhattan distance between metrics.

12.1 Why our algorithm doesn't work for hard challenges?

We have observed that when there is enough room for player to explore or when there are only few valid moves, our algorithm finds the solution quickly. For challenge problems, we have observed that most of the moves need to follow an order to get to the solution and there are limited number of possible paths to the solution. This means that we have to explore as much as possible and without strong heuristics it is hard to do so. We can try dynamically changing the step size for better exploration at the beginning but this should be corroborated by good heuristics.

13 Conclusion

We have successfully implemented Sokoban Solver using Q-learning. Our algorithm is able to solve the challenge test cases sokoban-01 to sokoban-05b.txt. We also tested our code against some custom Sokoban boards up to size 20x20 [See appendix A.2 and source code]. But for boards of size 10x10 and above with higher difficulty, our code fails to find a solution within an optimal time. For these boards, we could probably try to dynamically increase the step size to make the player reach the states that are not explored yet as typically for harder Sokoban boards, we have to use all the available states at our disposal to come to a solution. The ϵ-greedy approach without dynamic step size may take lot of time to explore important distant states. Also as stated in the limitations, we have to improve and add more heuristics. We can also try to use some other algorithm like A^* or IDS to compute initial heuristics value and pass them later to Q-learning.

14 Code link

https://github.com/hariuserx/Sokoban
How to run the code :
Simply change the input file name in the *sokoban_solver.py* at Line
and run *python3 sokoban_solver.py*

References

[1] https://en.wikipedia.org/wiki/Q-learning

[2] https://en.wikipedia.org/wiki/Sokoban

[3] Hasselt, Hado van - Reinforcement Learning in Continuous State and Action Spaces.

[4] Junghanns, Andreas; Schaeffer, Jonathan (1997). "Sokoban: A Challenging Single-Agent Search Problem" (PDF). In IJCAI Workshop on Using Games as an Experimental Testbed for AI Research. University of Alberta. pp. 27–36.

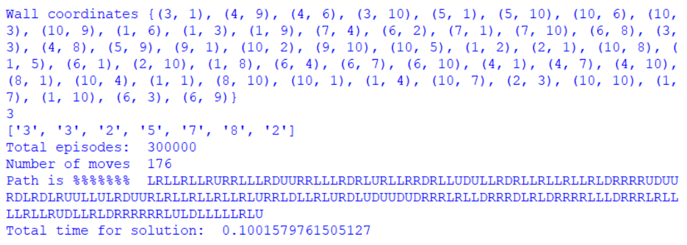
A Appendix

A.1 Pseudocode used for our implementation of Sokoban:

Time complexity is give for each method. Space complexity is also shown for QValues.
Overall Time complexity: $O(NumberOfBoxes * Length * Breadth * 4 * maxSteps * maxEpisodes)$ or until the time out of 60 min.
Overall Worst case Space complexity: $O((Length * Breadth)^{NumberOfBoxes}) * 4)$ We can only store top N highest Qvalues setting others to a small default value after some episodes to prevent high memory usage.

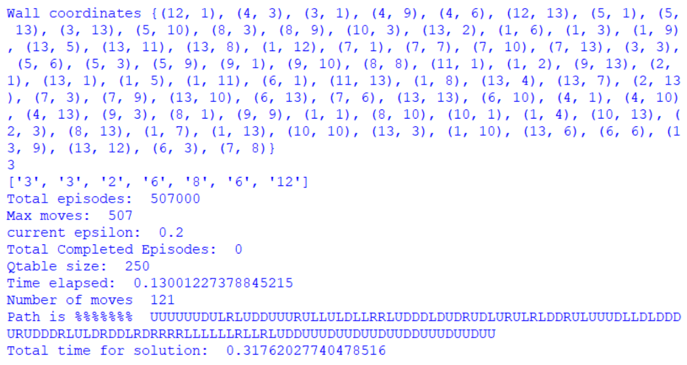
```
# return true if location is not a wall or box
#     otherwise return false
# Time complexity O(1)
def checkMovable(location, State):
    returns location not in State.walls and not in State.boxes

def bfs_getPath(State, location):
    # backtrack from location to an adjacent visited cell to get the path.
    path = []
    current_location = location
    for each adjacent cell C of current_location and C != location:
        if C is equal to 4:
            C = 5 # keep track of backtracking
            path.add([C.x, C.y])
            current_location = C

    return path.reverse()

# Time complexity O(N*M)
def bfs_checkIfPathExists(State, player_location, location):
    Queue queue
    queue.add(player_location)

    # let State here is a matrix and a duplicate copy,
    # denoting 1 for wall and 2 for box, 3 for empty cell.
    # 4 for visited
    while (queue is not empty):
        x,y = queue.pop()

        if (State[x][y] == 3):
            # set State[x][y] as visited, so we don’t visit it again
            State[x][y] = 4

        for (int i = -1; i <= 1; i+=2):
            if the coordinate (x + i, y) is (location.x, location.y)
                return bfs_getPath(State, player_location)
            if  the coordinate (x, y + i) is (location.x, location.y)
                return bfs_getPath(State, player_location)
            # if it is empty cell continue
            if State[x+i][y] == 3
                queue.add([x+i, y])
            if State[x][y+i] == 3
                queue.add([x+i, y])

    # Return NA if there is no path
    return "NA"
# Idea is to do BFS from player position to any of the adjacent
# cell of the box
# Time complexity O(N*M*4)
def locationsWherePlayerCanMoveAdjacentToBox(State, player_location, box_location):
    # Box location + {(-1, 0), (1, 0), (0, -1), (0, 1)}
    ans = []
    path = ""
    for each adjacent location of box_location:
        if (adjacent_location is empty cell AND (path =
              bfs_checkIfPathExists(State, player_location, location)) != "NA"):
            ans.add([location, path])

    return ans

### Q -learning
discount_factor = 0.8
directions = ['U', 'D', 'L', 'R']
learning_rate = 0.8

### After parsing the input file, store dimensions, wall
### locations, box locations, player location
### and storage locations,

## Initial locations
BoxLocations
PlayerInitialLocation
StorageLocations
Length
Breadth
WallLocations

## Only store BoxLocations in state as others are fixed and
## we are getting rid of PlayerLocation using BFS
class State
    # index each box for convenience
    Boxes = {index1 : BoxLocation1, index2 : BoxLocation2, ....}

### Utility methods
## Time complexity O(1)
def checkIfDeadlock(BoxLocation):
    if both edges at any of the 4 corners of the box are walls:
        return true
    return false

## Time complexity O(Number of boxes)
def checkIfTerminal(State)
    if the set itersection of BoxLocations and StorageLocations is equal to BoxLocations:
            return true
    return false

## Time complexity O(1)
def checkIfBoxCanBeMoved(State, Box, action):
    if the nextLocation derived from action is not another box or wall:
        return true
    return false

## Time complexity O(Number of boxes * Length * Breadth * 4)
def getAllValidActions(State, current_player_location):

    all_actions = []
    for each box B:
        paths = locationsWherePlayerCanMoveAdjacentToBox(State,
                                    current_player_location, B)
        for each path P:
            nextPlayerLocation = P.last()
            action = # Since player can only push the box we
                     # only check one direction where player is facing the box
            if checkIfBoxCanBeMoved(State, B, nextPlayerLocation)
                all_actions.add([action, path])

# Space complexity O((Length * Breadth)^(Number of boxes) * 4).
# We can only store top N highest qvalues setting others
# to a small default value after some episodes to prevent
# high memory usage.
QValues = {} # An empty dict/Map initially storing State,action - qvalue mapping.
# Initially all qvalues are 0.

## Time complexity O(Number of boxes * Length * Breadth * 4)
def getBestAction(State, player_location):
    actions = getAllValidActions(State, current_player_location)
    return the action A for which QValues.get([State, A]) is the maximum

## Time complexity O(Number of boxes)
def checkIfNoSolution(State):
    if any of box satisfies checkIfDeadlock(BoxLocation):
        return true
    return false

## Time complexity O(Number of boxes)
def getQValue(State, action):
    if checkIfNoSolution(State):
        return INT_MIN
    if checkIfTerminal(State):
        return INT_MAX

    return QValues.get([State, action])

## Time complexity O(Number of boxes * Length * Breadth * 4)
def performAction(State, BoxLocation, action, current_player_location):
    # action is a valid action here
    #newState = new state after the change of box location
    newState = State.copy()
    newBoxLocation = change BoxLocation coordinates based on action [Up, Right, Down or Left]
    newState.boxes[index_of BoxLocation] = newBoxLocation

    # Following are the scenarios. The reward values are just for reference.
    Reward = 0
    if BoxLocation to newBoxLocation is
            Storage to Storage: Reward += -10
            Storage to non-Storage: Reward += -100
            Non-storage to non-storage: Reward += -10
            Moving closer to the nearest storage location:  Reward += +10
            Moving farther from the nearest storage location:  Reward += -10
            Non-storage to storage:  Reward += +100

    currentQValue = getQValue([State, action])
    # Q value update
    newQValue = currentQValue + learning_rate * (Reward +
                                discount_factor *
                                getQValue(newState, getBestAction(newState,
                                    current_player_location))
                                - currentQValue)


    QValues.set([State, action]) = newQValue
    State = newState

# A episode is run until max configured steps are reached
## Time complexity O(Number of boxes * Length * Breadth * 4 * maxSteps)
def runEpisode():
    state = State(BoxLocations) # create a new state object
    maxSteps = Length*Breadth*BoxLocations.count()*4
    while(maxSteps-- > 0):
            if random < 0.3 # epsilion greedy
                action,path = random(getAllValidActions(State,  PlayerInitialLocation))
            else:
                action,path = getBestAction(State, PlayerInitialLocation)
            performAction(State, BoxLocation from action, action, path.last())
            path += action

            if  checkIfTerminal(state):
                return path
            if  checkIfNoSolution(state):
                return

# Main driver code
# Run for at max 60 min (configurable)
# Time complexity O(Number of boxes * Length * Breadth * 4 * maxSteps * maxEpisodes)
# or until 60 minutes
def runAlgorithm():
    # maximum number of episode we want to run
    maxEpisodes = 20000
    path = "NA"

    while (maxEpisodes-- > 0 AND timeElapsed < 60 minutes):
        path = runEpisode()
        if path != "NA":
            return path

    return path
```

A.2 Custom Input Solutions



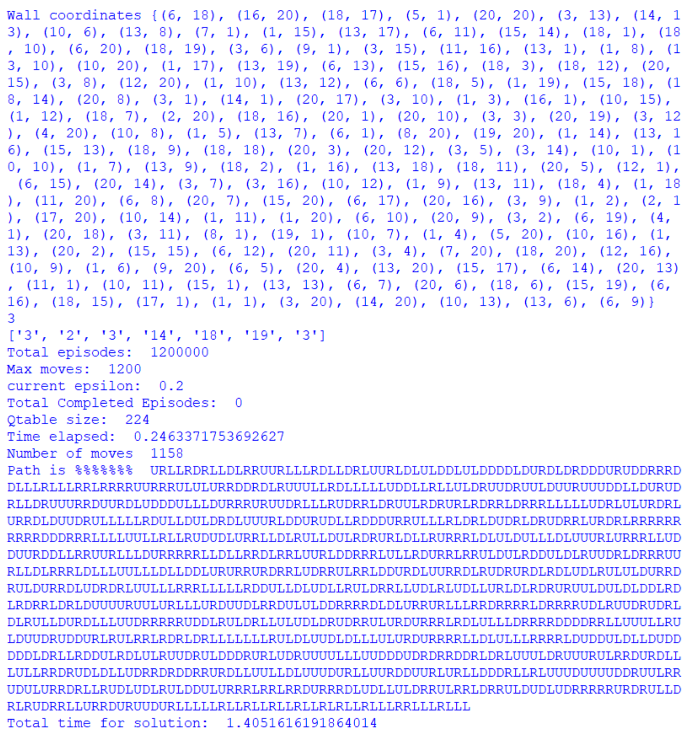
```
Wall coordinates {(3, 1), (4, 9), (4, 6), (3, 10), (5, 1), (5, 10), (10, 6), (10,
3), (10, 9), (1, 6), (1, 3), (1, 9), (7, 4), (6, 2), (7, 1), (7, 10), (6, 8), (3,
3), (4, 8), (5, 9), (9, 1), (10, 2), (9, 10), (10, 5), (1, 2), (2, 1), (10, 8), (
1, 5), (6, 1), (2, 10), (1, 8), (6, 4), (6, 7), (6, 10), (4, 1), (4, 7), (4, 10),
(8, 1), (10, 4), (1, 1), (8, 10), (10, 1), (1, 4), (10, 7), (2, 3), (10, 10), (1,
7), (1, 10), (6, 3), (6, 9)}
3
['3', '3', '2', '5', '7', '8', '2']
Total episodes:  300000
Number of moves  176
Path is %%%%%%%  LRLLRLLRURRLLLRDUURRLLLRDRLURLLRRDRLLUDULLRDRLLRLLRLLRLDRRRRUDUU
RDLRDLRUULLULRDUURLRLLRLLRLLRLURRLDLLRLURDLUDUUDUDRRRLRLLDRRRDLRLDRRRRLLLDRRRLRLL
LLRLLRUDLLRLDRRRRRRLULDLLLLLRLU
Total time for solution:  0.1001579761505127
```

Figure 16: Custom input: 10x10 sokoban



```
Wall coordinates {(12, 1), (4, 3), (3, 1), (4, 9), (4, 6), (12, 13), (5, 1), (5,
 13), (3, 13), (5, 10), (8, 3), (8, 9), (10, 3), (13, 2), (1, 6), (1, 3), (1, 9)
, (13, 5), (13, 11), (13, 8), (1, 12), (7, 1), (7, 7), (7, 10), (7, 13), (3, 3),
 (5, 6), (5, 3), (5, 9), (9, 1), (9, 10), (8, 8), (11, 1), (1, 2), (9, 13), (2,
1), (13, 1), (1, 5), (1, 11), (6, 1), (11, 13), (1, 8), (13, 4), (13, 7), (2, 13
), (7, 3), (7, 9), (13, 10), (6, 13), (7, 6), (13, 13), (6, 10), (4, 1), (4, 10)
, (4, 13), (9, 3), (8, 1), (9, 9), (1, 1), (8, 10), (10, 1), (1, 4), (10, 13), (
2, 3), (8, 13), (1, 7), (1, 13), (10, 10), (13, 3), (1, 10), (13, 6), (6, 6), (1
3, 9), (13, 12), (6, 3), (7, 8)}
3
['3', '3', '2', '6', '8', '6', '12']
Total episodes:  507000
Max moves:  507
current epsilon:  0.2
Total Completed Episodes:  0
Qtable size:  250
Time elapsed:  0.13001227378845215
Number of moves  121
Path is %%%%%%%  UUUUUUDULRLUDDUUURULLULDLLRRLUDDDLDUDRUDLURULRLDDRULUUUDLLDLDDD
URUDDDRLULDRDDLRDRRRRLLLLLLRLLRLUDDUUUDUUDUUDUUDDUUUDUUDUU
Total time for solution:  0.31762027740478516
```

Figure 17: Custom input: 13x13 sokoban



```
Wall coordinates {(6, 18), (16, 20), (18, 17), (5, 1), (20, 20), (3, 13), (14, 1
3), (10, 6), (13, 8), (7, 1), (1, 15), (13, 17), (6, 11), (15, 14), (18, 1), (18
, 10), (6, 20), (18, 19), (3, 6), (9, 1), (3, 15), (11, 16), (13, 1), (1, 8), (1
3, 10), (10, 20), (1, 17), (13, 19), (6, 13), (15, 16), (18, 3), (18, 12), (20,
15), (3, 8), (12, 20), (1, 10), (13, 12), (6, 6), (18, 5), (1, 19), (15, 18), (1
8, 14), (20, 8), (3, 1), (14, 1), (20, 17), (3, 10), (1, 3), (16, 1), (10, 15),
(1, 12), (18, 7), (2, 20), (18, 16), (20, 1), (20, 10), (3, 3), (20, 19), (3, 12
), (4, 20), (10, 8), (1, 5), (13, 7), (6, 1), (8, 20), (19, 20), (1, 14), (13, 1
6), (15, 13), (18, 9), (18, 18), (20, 3), (20, 12), (3, 5), (3, 14), (10, 1), (1
0, 10), (1, 7), (13, 9), (18, 2), (1, 16), (13, 18), (18, 11), (20, 5), (12, 1),
 (6, 15), (20, 14), (3, 7), (3, 16), (10, 12), (1, 9), (13, 11), (18, 4), (1, 18
), (11, 20), (6, 8), (20, 7), (15, 20), (6, 17), (20, 16), (3, 9), (1, 2), (2, 1
), (17, 20), (10, 14), (1, 11), (1, 20), (6, 10), (20, 9), (3, 2), (6, 19), (4,
1), (20, 18), (3, 11), (8, 1), (19, 1), (10, 7), (1, 4), (5, 20), (10, 16), (1,
13), (20, 2), (15, 15), (6, 12), (20, 11), (3, 4), (7, 20), (18, 20), (12, 16),
(10, 9), (1, 6), (9, 20), (6, 5), (20, 4), (13, 20), (15, 17), (6, 14), (20, 13)
, (11, 1), (10, 11), (15, 1), (13, 13), (6, 7), (20, 6), (18, 6), (15, 19), (6,
16), (18, 15), (17, 1), (1, 1), (3, 20), (14, 20), (10, 13), (13, 6), (6, 9)}
3
['3', '2', '3', '14', '18', '19', '3']
Total episodes:  1200000
Max moves:  1200
current epsilon:  0.2
Total Completed Episodes:  0
Qtable size:  224
Time elapsed:  0.2463371753692627
Number of moves  1158
Path is %%%%%%%  URLLRDRLLDLRRUURLLLRDLLDRLUURLDLULDDLULDDDDLDURDLDRDDDURUDDRRRD
DLLLRLLLRRLRRRRUURRRULULURRDDRDLRUUULLRDLLLLLUDDLLRLLULDRUUDRUULDUURUUUDDLLDURUD
RLLDRUUURRDUURDLUDDDULLLDURRRURUUDRLLLRUDRRLDRUULRDRURLRDRRLDRRRLLLLLUDRLULURDRL
URRDLDUUDRULLLLLRDULLDULDRDLUUURLDDURUDLLRDDDURRULLLRLDRLDUDRLDRUDRRLURDRLRRRRRR
RRRRDDDRRRLLLLUULLRLLRUDUDLURRLLDLRULLDULRDRURLDLLRURRRLDLULDULLLDLUUURLURRRLLUD
DUURDDLLRRUURLLLDURRRRRLLDLLRRDLRRLUURLDDRRRLULLRDURRLRRULDULRDDULDLRUUDRLDRRRUU
RLLDLRRRLDLLLUULLLDLLDDLURURRURDRRLUDRRULRRLDDURDLUURRDLRUDRURDLRDLUDLRULULDURRD
RULDURRDLUDRDRLUULLLRRRLLLLLRDDULLDLUDLLRULDRRLLUDLRLUDLLURLDLRDRURUULDULDLDDLRD
LRDRRLDRLDUUUURUULURLLLURDUUDLRRDULULDDRRRRDLDLURRURLLLRRDRRRRLDRRRRUDLRUUDRUDRL
DLRULLDURDLLLUUDRRRRRUDDLRULDRLLULUDLDRUDRRULURDURRRLRDLULLLDRRRRDDDDRRLLUUULLRU
LDUUDRUDDURLRULRRLRDRLDRLLLLLLLRULDLUUDLDLLLULURDURRRRLLDLULLLRRRRLDUDDULDLLDUDD
DDDLDRLLRDDULRDLULRUUDRULDDDRURLUDRUUUULLLUUDDDUDRDRRDDRLDRLUUULDRUUURULRRDURDLL
LULLRRDRUDLDLLUDRRDRDDRRURDLLUULLDLUUUDURLLUURDDUURLURLLDDDRLLRLUUUDUUUUDDRUULRR
UDULURRDRLLRUDLUDLRULDDULURRRLRRLRRDURRRDLUDLLULDRRULRRLDRRULDUDLUDRRRRRURDRULLD
RLRUDRRLLURRDURUUDURLLLLLRLLRLLRLLRLLRLRLLRLLLRRLLLRLLL
Total time for solution:  1.4051616191864014
```

Figure 18: Custom input: 20x20 sokoban