CS271 Final Project
University of California, Irvine

THE SOKOBAN GAME REINFORCEMENT LEARNING

December 10, 2021

Ashley Schwartz
Anand Srinivasan
Erik Urzua

Contents

1 Introduction

1.1 Sokoban Game

Sokoban is a puzzle game originally invented by Hiroyuki Imabayashi in 1982, where a player controls a character and is to move various boxes around a two dimensional grid of squares until all the boxes are placed on an equal number of goal/storage locations [1][1]. In addition to the boxes and storage locations, the grid is also filled with and surrounded by walls that the character nor a box can move through. This creates a maze-like grid world. The character can only move vertically (up and down) and horizontally (left and right) into an empty square or storage location and move one single box into an empty square or storage location by "walking" up to the box and moving in the direction of said empty square or storage location. With these rules the player is unable to "pull" a box nor move two boxes in at a time. To see the solution path of a simple/trivial Sokoban puzzle, see Figure 1[2].

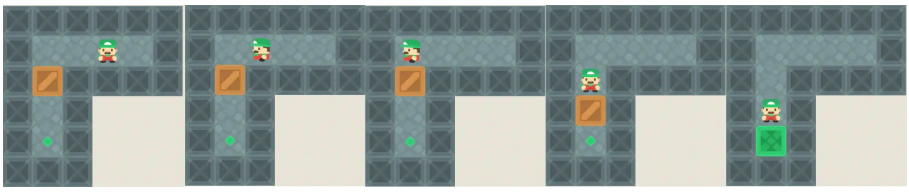


Figure 1: Screen grabs of a simple Sokoban puzzle that demonstrates the movements to solve the puzzle. The leftmost screen grab is the initial state of the game and the rightmost is the end state when the lone box has been "placed" and or "pushed" onto the only storage location.

Although Sokoban is a fairly easy game to describe and conceptually understand given its set of rules, it is in fact quite a difficult game to solve for both human and artificial intelligence (AI) players. Not only is solving a Sokoban puzzle a NP-hard problem [2], but it has also been shown that solving a Sokoban problem and other push-pull problem is PSPACE-complete[3, 4].

The difficulty in solving the Sokoban puzzles inherently lay in the brief and simple game rules. Most games require a large number of tangling, crisscrossing and backtracking moves by the player/character to push all the boxes onto all the storage locations, for which many of the move are irreversible and can result in a **deadlock** state as the boxes can only be pushed. As such, most Sokoban games have large search spaces that are intractable to standard search based solution methods. As an example, in the popular Sokoban puzzle set XSokoban, commonly used in AI research, the puzzles have a branching factor as high as 136 with an average of 12, a solution length between 97 and 674 moves with an average of 260 and an upper bound on the search-space size of 10^{98} with a median size of 10^{18} [5, 6] though. Deadlock states are game states for which the game cannot be solved due to the existence of irreversible moves and that boxes can only be pushed. Deadlock states differ from any irreversible move in that an irreversible move may certainly restrict which boxes can now be placed on a certain or multiple storage locations, but the game can still be solved after that irreversible move have been made. Examples of deadlock states can seen in Figure 2 below, though there are many more deadlock states then these few shown. The difficulty presented with deadlock states is in detecting if a move made by the player/character results in a deadlock state as determining this amounts to finding a solution for the puzzle from that state.

[1]We note that since the original creation, many different versions and implementation of the Sokoban puzzle game have been created where the objects to be moved and placed on goal states are not boxes but rather stones, crates, boulders and other heavy and difficult to move objects. Throughout the rest of this paper though we will refer to the objects that are to be moved onto the goal state(s) (or storage locations) as boxes since the Japanese word "sokoban" roughly translates to warehouse man.

[2]This puzzle can be found in the free browser-based Sokoban version for children created by the Canadian Broadcasting Company and is accessible here.

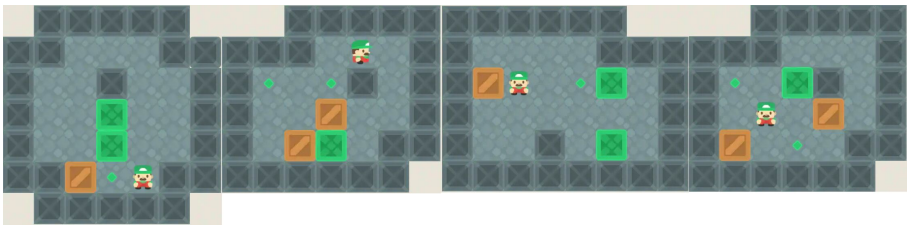


Figure 2: Four example deadlock states, two of which come from the same puzzle. Note that even though the game is in a deadlock other blocks can still be moved around.

This project aims to solve the Sokoban game using AI techniques. As described previously, the state-space size for the problem can become increasingly large. This means the technique used to solve the problem must be carefully chosen, as many will fail due to computational and time restrictions. Additionally, the game itself is largely dependent on the player's moves, as its moves will change the environment and state space greatly with moving boxes being a key component. Reinforcement learning is a methodology perfect for this problem as we can use the player's experiences in the game to teach the player how to win the game. In AI and throughout this project, we refer to a player as an agent.

1.2 Reinforcement Learning

Reinforcement learning is a methodology in which an agent interacts with an environment and receives rewards for doing so. A reward in the context of Sokoban can be putting a box on a goal, or putting all boxes on goals. It is based largely on a defined Markov Decision Process (MDP) that defines the world and its rewards based on certain actions. It differs from simply solving a MDP because the agent is not given the MDP itself. In fact, it aims to learn the MDP as it interacts with the environment repeatedly. In a basic MDP where a perfect model of the environment exists, we can compute the optimal policy using dynamic programming (DP). DP uses iterative algorithms such as value iteration and policy iteration that are based on the Bellman optimality equations. While it is possible to solve the Sokoban problem using value or policy iteration, the exponential space requirement [7] for grid sizes makes it impractical in most cases. These ideas are wrapped into temporal-difference (TD) learning, for which we will use to solve the Sokoban game.

There are a variety of different ways to implement TD learning such as on-policy SARSA and off-policy q-learning.In this project we will use off-policy TD control q-learning. Specifically, we are using model-free reinforcement learning technique for which we are using action-utility learning. The agent is going to learn a Q-function that denotes the sum of rewards from state s onward if action a is taken.

2 Problem Development

In the following sections we define the design of the problem itself and how to solve the game of Sokoban. We begin by setting the stage for the problem itself through problem development, discuss the AI techniques used in algorithm form, and finally discuss the data structures needed to solve Sokoban. Definitions and topics in this section are adopted from [8].

2.1 The Sokoban Game State

The fundamental idea of reinforcement learning is to maximize the sum of rewards. A MDP and its components, although not solved directly in reinforcement learning, is how we will define the

components of the problem. The components are a set of states, a set of actions, and a reward function. These components are described in detail along with their computational attributes.

The implementation of the Sokoban game is performed on a rectangular grid of size $m \times n$, where m and n are integers representing the height and width of the board. Each cell of the board is a **state** s, with initial state s_0. The cells, or states, are further classified as valid and invalid based on the occupancy status. As an example, the locations of walls cannot be accessed and are designated accordingly.

The location of the **agent**, or AI player, at any instant is identified by its location on the board. The allowable set of **actions** for the agent from one cell to the next is designated as the set

$$Actions(s) = \{Up, Down, Left, Right\}.$$

Allowable actions are then adjusted accordingly at each step based on the locations of walls (ie., locations that are unreachable). In addition to the unreachable states from a wall, the position of a box with respect to the locations of walls can render certain actions as invalid. One such example is the location of a wall to the right of the location of a box, which would render the right-action as impossible for the agent.

The **terminal goal state** is said to be achieved when all boxes are in the same cells as each of the goal-cells. It is therefore natural that the number of boxes equal the number of cells with goals in a valid Sokoban level. It should be underscored here that the agent and the boxes are movable objects (with coordinates that get updated continually during the course of the game), while the locations of the walls and the goals remain fixed.

The **reward** function, denoted $R(s, a, s')$, is originally designated as described by Equation 1. We also have a discount factor γ, a number between 0 and 1 that determines the models factor towards future rewards, and a learning rate parameter α, a number between 0 and 1 that defines the agents ability to learn from new actions.

If we set γ to zero, the agent completely ignores the future rewards.

$$R(s, a, s') = \begin{cases} +1 & \text{all boxes on goals} \\ +0.1 & \text{box pushed on goal} \\ -1 & \text{box pushed to deadlock} \\ -0.1 & \text{all other moves} \end{cases} \tag{1}$$

The largest difference from a full MDP and reinforcement learning is the transition model is not known. The transition model, although never used in the computations of reinforcement learning, is still worth defining as it is what we aim to approximate. We will also often talk about the notion of a utility that is associated with a transition model in a MDP.

2.2 Q-learning Preliminaries

For this project we aim to implement q-learning, which is an off-policy TD control. We implement active reinforcement learning for which an agent gets to decide which actions to take. The agent is an active learning agent (ADP). In ADP, the agent needs to learn a complete transition model with outcome probabilities for all actions rather than those defined by a policy such as in passive dynamic programming. The transition model approximations are attempting to determine the utilities defined by the optimal policy that obey the Bellmann equations, Equation 2.

$$U(s) = \max_{a \in A(s)} \sum_{s'} P(s|s, a) \left[R(s, a, s') + \gamma U(s') \right] \tag{2}$$

The agent's goal is to learn the transition model and in turn, learn the best actions to take. To actually do the learning, the agent can interact with the environment in one of two ways: explore or exploit. We call each time an agent interacts with the environment a trial or an

episode. When an agent is in exploration mode during an episode, it takes actions at random. When an agent is in exploitation mode, it takes an action that it believes to be optimal, or in favor of high (estimated) utility values. How exactly the agent determines an optimal action is part of the fundamentals of q-learning and is the basis for a greedy agent, i.e., one that favors high utilities.

There must be a balance between exploring the environment and exploiting it and this is determined by an exploration function. We have chosen to take an ϵ-greedy strategy. In this case, we have a generated random number ρ for which we choose exploration when $\rho < \epsilon$ and exploitation when $\rho > \epsilon$ such that ϵ decreases for some finite value after each episode. This ensures the agent chooses to explore more in the beginning and then begins to exploit the environment as the of episodes number increases.

The Q-learning method learns an action-utility function $Q(s, a)$ instead of a utility function $U(s)$. The idea is that the agent is able to act optimally by choosing the action that yields maximum $Q(s, a)$. The updates of these values occur at each time step when a state, action pair is executed. The Q-learning TD update is shown in Equation 3 where α is the learning rate parameter defined by an exploration function as discussed previously.

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[R(s, a, s') + \gamma \max_{a'} Q(s', a') - Q(s, a) \right] \tag{3}$$

2.3 Feasibility of the Defined Problem

As described previously, the Sokoban game complexity lies largely in the expansive amount of states that can be reached. In Q-learning, we eliminate the need to define the state as an entire board and therefore, diminish the computational power needed. Another benefit to Q-learning is we assign rewards based off an observed state action pair and the whole environment is never needed to be known by the agent. As the agent explores the environment, it will learn the transition model and eventually determine a solution to the problem.

3 Computational Techniques

Q learning is an iterative process where the Q value, $Q(s, a)$, is calculated and updated at each time step. The agent interacts with the environment by exploration or exploitation for many episodes until the agent ideally learns the optimal path and solves the game. In this section we describe how exactly we do this computationally along with the data we aim to hold.

3.1 Data Structures

The data structures that define the game are developed using a Python class. We define a Sokoban game that has attributes such as an agent, goals, boxes, and walls. A game board is stored using a two-dimensional array where its components reflect the current state of the board (i.e., where the boxes and agent currently is). The space complexity of the board is the specified size of $m \times n$. The object from the class is only referenced for environment checking, e.g. if boxes are on goals, and to keep track of the number of times a state action pair is visited.

We note there is not a specific reward for every state, but rather there is a reward for a state with an associated action. This is important as we are not required to store rewards and utilities for each state. Specifically, as boxes move in the environment, the reward of a state can change but a reward given an action (e.g., an action that moves a box) will stay the same.

The main data structure for the q-learning algorithm is the q-table that is updated at each time step. The table is size $x \times y$ where x is the number of actions and y is the number of states. Each element is initialized to zero and is updated only when the ADP reaches that state given a specific action. An example of a q-table for the problem is shown in Table 1. This means the

space complexity here is $x \times y$. This is simply stored in a two-dimensional list using the numpy package in python.

	$\Uparrow$	$\Downarrow$	$\Leftarrow$	$\Rightarrow$
s_0	0	0	0	0
s_1	0	0	0	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
s_m	0	0	0	0

Table 1: Initial q-table for the sokoban game where the elements, initially zero, are the q-value $Q(s, a)$

We also keep track of the frequencies of the state-action pairs. This is a table of identical size of the q-table for which its elements are the frequency of visiting that state action pair. The space complexity is $x \times y$.

The data storage for the problem is mainly dictated by the number of states (board cells) and number of actions. If N_s is the number of states and N_a is the number of actions, the necessary space is $2N_sN_a$.

3.2 Algorithm

The Q-learning algorithm is executed for a multitude of episodes for a multitude of time steps per episode. We require a maximum bound for the number of trials and number of time steps. We may view the upper bound of time steps as a penalty to the agent for not solving the game within a specified amount of time. The algorithm, comprised mainly of the q-update, is shown in Algorithm 1.

Algorithm 1 Q-learning-algorithm

Require: number of episodes, number of time steps, MDP
  for each episode **do**
    $timestep \leftarrow 0$
    while s not terminal, timestep $<$ max timesteps **do**
      Choose $a \leftarrow \epsilon\_greedy(\epsilon)$
      Take action a, observe r, s'
      $Q[s, a] \leftarrow Q[s, a] + \alpha(N_{sa}[s, a])(r + \gamma \max_{a'} Q[s', a'] - Q[s, a])$
      $s \leftarrow s'$
      $timestep \leftarrow timestep + 1$
    end while
  end for

A key component is terminal state checking. An episode only completes if the agent has reached a terminal state (or if the maximum number of time steps has been reached). A terminal can occur if the agent has put all boxes on goals, or if the agent is in a deadlock, Algorithm 2.

Algorithm 2 Is-Terminal

Require: MDP
  if deadlock or box on each goal **then**
    return True
  else
    return False
  end if

3.3 Deadlock Checking

As was briefly mentioned above in Section 3.3, deadlocks are states of the games board such that a solution to the puzzle cannot be found from this state. In general, determining that any Sokoban puzzle board is in a deadlock is as difficult as finding the solution to the puzzle from that state as finding that there is no solution is the complement of finding a solution. However there are specific types of deadlock that can be easily checked for in any state of any Sokoban board and these types are the ones that we check for while the agent attempts to solve any Sokoban puzzle. The first of these simple types of deadlock is when a box is located in a "corner" and is not already placed on a goal where a corner is any open/empty grid cell that has two neighboring wall cells that are not situated on opposite sides of the open/empty cell. An example of a box placed in a corner and not on a goal square can be seen in the leftmost and rightmost games boards of Figure 2. Any board with a single box located in such a position is in a deadlock state since the player cannot move the square from that position and thus there will always be one goal that cannot have a box placed upon it.

The second type of deadlock checking that we preform is to check if, for any box that is flush up against a series of walls, that that box can be moved onto a goal. These types of boxes are such that on at least one side of the box, b, there is a wall, w, and that for every grid cell that the box can be moved in a direction orthogonal to its neighboring wall w, there is also a wall located on the same side as w was to the box b. An example of such a box can be seen in the second Sokoban game puzzle from the right in Figure 2. For such a box, it can only possibly be moved vertically, up or down, should the box have a wall located to its right or left or horizontally, left and right, if the box has a wall located above or below it. Thus if there is no goal cells reachable, the box can never be placed on a goal and so the game is in a deadlock state. It should also be mentioned that in the process of determining if a given Sokoban game board is in this type of deadlock we also check that the number of goals along the series of walls that the box b is flush up against is less than or equal to the number of boxes flush up against the walls. If the number of goal cells is less than then the number of boxes against the walls than the board is in a deadlock. For this reason, when we check if a box, b, is flush up against a series of walls as previously described, prior to determining if the box in question can reach a goal we first check to see if the number of boxes along the walls is more than the number of goals along these wall as there being less goals than boxes means that for one of the boxes, it is unable to reach a goal. Should the number of goals be greater than or equal to the number of boxes, then we check for the box b, that it can reach a goal by being pushed alongside the walls to the goal. If it cannot be pushed to some goal then the current state of the Sokoban puzzle is a deadlock.

The third and final type of deadlock that we check for is a grouping of boxes and walls such that none of the boxes that are already on a goal in the group can be moved. An example of this is that for a box b located at the (i, j) grid cell, there is also a box located at grid cells $(i+1, j)$, $(i, j+1)$, and $(i+1, j+1)$ (below, to the right, and lower right diagonal positions relative to b respectively). In this example none of the four boxes can be moved since the player/character in the game cannot move two boxes at once. In generally though any three of the four boxes in the example can be replaced by a wall and still the grouping of the boxes and walls are such that none of the boxes can be moved. Another example with two boxes is when both boxes have a

wall on their right and where either one of the two boxes is above the other. To check for this type of deadlock state, for each box b currently not on a goal and located at grid cell (i_b, j_b), we look to see if there are walls and or other boxes at the eight grid points, $(i_b + k, j_b + l)$ for $k = -1, 0, 1$ and $l = -1, 0, 1$ with k and l both not being 0, that surround b. If so and if the surrounding walls and boxes of b create a pattern similar to that exemplified in two examples then the game is in a deadlock state.

The process and algorithm that we use for checking if a given Sokoban game is in a deadlock state is seen in Algorithm 3 below.

Algorithm 3 dead_lock

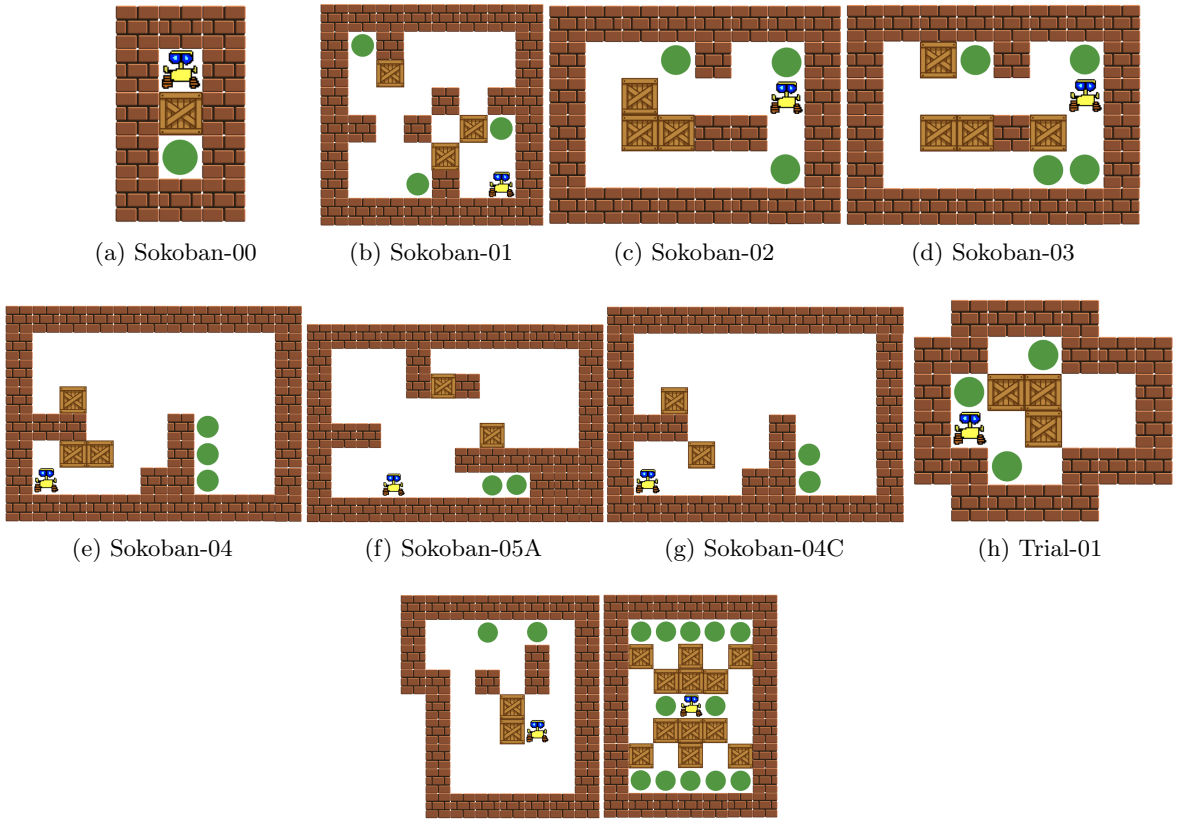
Require: Board of Sokoban game
  for each box b in the board **do**
    if b is in a corner and not on a goal **then**
      return True
    end if
    if b is flush along wall and no reachable **then**
      return True
    end if
    if b is in a group **then**
      return True
    end if
  end for
  return False

4 Results

We have implemented our Q-learning algorithm on a variety of Sokoban boards as shown in Figure 3 with performance metrics shown in Table 2. Most boards were tested using 1000 episodes and around 500 maximum steps allowed per episode. The discount factor γ was chosen to be 0.99, the learning rate α was chosen to be 0.01, and the exploration parameter *epsilon* was chosen to be 0.1. In the case described, the agent more often exploits than explores, learns slowly, and looks for high rewards in the future. The algorithm was able to converge to the optimal solution in some instances, but often times did not achieve optimally within the allotted episode and step allowances.

Sokoban Board	Solved	Trial Number	Number of Steps (first solution found)	Number of Steps (final solution found)
00	yes	1	1	1
-01	yes	7	72	35
-02	yes	108	307	307
-03	yes	122	162	150
-04	no	X	X	X
-05a	yes	10	359	85
(G)	yes	16	161	161
(H)	yes	6	83	11
(I)	yes	3	103	21
(J)	yes	908	812	812

Table 2: Solution results



(a) Sokoban-00 (b) Sokoban-01 (c) Sokoban-02 (d) Sokoban-03

(e) Sokoban-04 (f) Sokoban-05A (g) Sokoban-04C (h) Trial-01

(i) Trial-02 (j) Trial-03

Figure 3: Sokoban boards used for testing algorithm performance.

5 Performance Observations

The environment in Sokoban is ever changing and often times difficult to manage. The first factor we noticed in our implementation that posed a road block was differentiating between an actions such as normal moves and pushing a box and managing the aspect of time. Regardless of where an agent is located within a board, its move at one time point will need often to be different from another. In Q-learning, the agent doesn't know anything about the entire environment around it and only knows what rewards it may get from executing an action. This sometimes causes issues when an agent needs to execute different actions from the same state at different points in time. We found that this factor played a large role in causing our program to fail for complicated boards where an agent revisits a board location multiple times.

Aside from the road block discussed above, there were other noticeable factors that contributed largely to an agent finding a solution. The observed problems include deadlock checking, learning rate, exploration parameter, and reward values.

The most noticeable challenge for the agent to solve a game is the threat of a deadlock. We have implemented a variety of deadlock checkers but there are cases where the agent finds itself in a board that is unsolvable. As this goes undetected, the agent tends to move in the board aimlessly, skewing its previous and future learning. To combat this, we have set a maximum number of steps an agent is allowed to take within an episode before it is forced to start over. Even with this safeguard, it is hard to identify an optimal upper bound for the number of steps as early learning often takes large amounts of steps. As an example, in the trial for Sokoban board 02 the agent found a solution in 307 steps. This board was particularly tricky because if the agent moved a box into the bottom row, the board is immediately unsolvable but it not viewed as a classical deadlock since there is a goal state within that row.

The learning rate parameter α was very sensitive for solution finding. In a simple board like (h), for example, the agent can find a solution within 7 episodes and took 72 steps. When the learning rate is changed from 0.01 to 0.7, the agent doesn't find a solution until episode 515 for which it takes 165 steps. In some boards, if α was not chosen properly, the solution was never found within a maximum of 1000 episodes.

Whether or not the agent chooses to exploit or explore the board played a large role in whether or not it gets "stuck". If the agent consistently exploits the board, there is no room for the agent to try new things and determine if a different move would be better. On the other hand, if the agent consistently explores, it fails to reinforce good behavior. We have chosen ϵ to be 0.1, meaning there is a 10% chance the agent will explore for any specific move. In some trials, we noticed the agent was having trouble revisiting states and wanting to execute previous actions without reaching other areas of the board. In these instances, a higher ϵ was better as the chance for the agent to explore increased.

In different boards, an agent might have to move a box a long way to reach a goal. In turn, it was sometimes beneficial to reward an agent for moving a box, even incrementally. The choice of this reward is crucial. If you reward an agent too much, it learns to move a box back and forth and never tries to reach a goal. In the case where a goal state was far away, we often chose a reward value of -0.01, which is a smaller penalty than normal moves, for moving a box. This was helpful for Sokoban 05A for which we wanted to incentivize the agent to move a box a long way.

6 Conclusion and Future Directions

The Q-learning algorithm seemed to perform well in solving the Sokoban game. As it is known, the environment is ever changing and so fully observable methods would take a lot of computational power to solve a game. In Q-learning, we are able to overcome this hurdle, but at a cost. Since the agent doesn't know its entire environment, it has difficulty remembering

what it did in the past and how that is different than what it would like to do in the present. It would be interesting to explore different ways to help the agent change its actions based on if it has visited that state within the same episode. This could be done by tracking state frequency and directing the agent to execute an action based on how many times it has visited that state already.

We may also include state visit frequency to improve other factors that we noticed hindering algorithm performance. We noticed the exploration parameter and the learning rate played large roles in solution finding. For this project, we implemented an $\epsilon - greedy$ strategy for which ϵ was constant. In the future, it could be worth changing ϵ based on the number of times an agent has visited the state, or the number of episodes it has executed. The learning rate parameter may also be chosen this way although choosing the way in which you alter these values is not straightforward.

We may also improve our algorithm performance in the future by checking deadlocks we have not already accounted for. When an agent gets stuck, this degrades performance greatly and hinders the agent from making sufficient progress. However most deadlocks are similar to the one mentioned above in the Sokoban-02 (c), where, by moving a box into some position it restricts the movement of the player, or agent, in such a way that the other boxes cannot be be placed on goals. The deadlock states that we check for here do not consider how the movement or movements of a box effect the other boxes and their ability to reach a goal. Rather our deadlock checking is almost exclusively concerned with a box being in a position that it is not able to reach a goal. The distinction here is that, our deadlock checking does not, in a sense, look into the future and see that some move by a player, and or agent, has caused a problem that will not be seen until many moves later. Therefore, any continuing work must utilize better deadlock checking techniques and methods than were used here as the overwhelming majority of deadlock states are of the former type than the ones we have checked for . One such idea would be to incorporate some sort of a depth limited future state space search method, with or without heuristics search strategies, in an attempt to allow the agent to "see" some steps ahead after moving a box much like a human player can see how a move will effect the game at later points in time. An idea such as this one would ideally create a hybrid algorithm that combines the strength of classical search techniques and reinforcement learning to produce results better than each would give alone and those found here.

References

[1] F. Van Lishout, "Single-player games: Introduction to a new solving method," Ph.D. dissertation, University of Liège, Liège, Belgium, 2006.

[2] D. Dor and U. Zwick, "Sokoban and other motion planning problems," *Computational Geometry*, vol. 13, no. 4, pp. 215–228, 1999.

[3] S. I. Pspace-complete, J. C. Culberson, and J. C. Culberson, *Sokoban is pspace-complete*, 1997.

[4] A. G. Pereira, M. Ritt, and L. S. Buriol, "Pull and pushpull are pspace-complete," *Theoretical Computer Science*, vol. 628, pp. 50–61, 2016.

[5] A. Junghanns and J. Schaeffer, "Sokoban: Enhancing general single-agent search methods using domain knowledge," *Artificial Intelligence*, vol. 129, no. 1-2, pp. 219–251, 2001.

[6] A. Junghanns, A. Junghanns, and M. Eltern, "Pushing the limits: New developments in single-agent search," Tech. Rep., 1999.

[7] V. Ge, "Solving planning problems with deep reinforcement learning and tree search," *Annalen der Physik*, vol. 322, no. 10, pp. 891–921, 1905. DOI: `http://dx.doi.org/10.1002/andp.19053221004`.

[8] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Prentice Hall, 2021.