CS271 Sokoban Final Report

Dheyay Desai	Xinyuan Lin	Qihong Chen
42319808	26642054	70798945

December 10, 2021

Contents

1 Problem Statement

In this project report, we will demonstrate a Reinforcement learning algorithm based on Q-learning to find an optimal solution for the game of Sokoban on different input and board sizes.

1.1 Sokoban

The Sokoban Game is played on a board of squares, where each square is a floor or a wall. On the board, some floor squares contain boxes and storage. The player is confined to the board and may move horizontally or vertically. The player cannot move through the walls of boxes and can move only one box at a time. The player is also restricted from moving or pushing boxes stacked together in any direction. The goal for the player is to place all the boxes at a storage location without ending up in a deadlock that cannot be resolved.

Sokoban is a complex problem to solve because of the constrained nature of the game. The dependencies of future moves on a recent decision are enormous. Furthermore, certain moves are irreversible, which means if a player moves a box into a position surrounded by other boxes or walls, that box is rendered useless. The game objective cannot be completed. Each situation leads the player to restart the game from the initial board and make the moves again to find an optimal solution.

2 Approach

2.1 Preparation

Before starting the AI algorithm itself, we decided to create a base framework for the game of Sokoban so it would be easier for us to visualize and understand how our algorithm would work in real-time. We modularized our project into the following parts:

- Game Board - Contains the game board and the underlying information for the game state and updated block and player positions.
- Graphic interface - Helps visualize the game setup and runs when training to show how the decision making process of the player is working in each training iteration.
- Training module - Trains the agent on the board using our designed Reinforcement learning algorithm.
- Evaluation module - Runs the game

2.2 Initial approach

The initial approach we designed used a basic Monte Carlo Tree Search algorithm that employed a UCT table for actions to guide the agent's decision-making. The algorithm adopted the epsilon-delta policy to collect episodes, and depending on the results, the UCT was updated, and the episode was terminated once the step size was reached. This approach also used positive rewards for pushing a box and additional rewards for pushing a box into a storage unit while giving a negative reward for each move made. However, this approach would not benefit us as there are multiple scenarios to be considered in the game, and multiple boxes need to be pushed to storage. More often, the agent could push a few boxes to storage. Still, the episode would terminate since the game was not won, and these beneficial moves were not updated into the UCT table, and the agent would make extra moves resulting in a worse actual reward.

This method did not work due to our limited knowledge in implementing a single-player game MCTS. The results also were not promising enough for us to continue with the said algorithm.

2.3 Final approach

Our revised approach was based on the idea of using Q-learning. Once the input has been parsed and a board is generated, we use a custom structure called the Bad State dictionary to mark spaces on the game board that the player might never want to push a box into. These states are the irreversible "deadlock" positions that are either surrounded by walls or in corners. We update this dictionary as moves are made since positions cannot be moved depending on the game board state and the position of boxes. This table helps the agent in its decision-making process, making it easier not to fall into a trap or a deadlock. This helps us reduce time to search for a deadlock in every iteration of the training algorithm and avoid wasting time over move sequences that lead to such situations. The Q-value table estimates how good/bad a movie is when selecting a move from possible moves. This table is generated during the agent's training on the game board using the Q-learning algorithm.

The main Q-learning algorithm follows this structure:

- Initialize Q table and state value table
- Use Breadth First search for all possible paths to movable boxes
- Determine move based on policy (greedy or random)
- Execute selected action and check terminal state, win or loss based on deadlocks or boxes moved to storage
- Determine reward based on move made
- Update Q-value table and state table for selected action
- Update game state and get updated paths
- Repeat until training step size exceeded

In the final run, the agent uses the already updated Q-table for decision making when selecting moves from any position in a non-deterministic manner, selecting a random move for breaking a tie between multiple moves with a similar Q value.

3 Algorithms

3.1 Q-learning

The pseudo code for the described Q-learning approach. The different functions/algorithms used here are mentioned and explained below.

Algorithm 1: Q-Learning Training

Input : BoardState
Input : ε - base epsilon value
Output: Saves Q Table values
$BoxesDone \leftarrow 0$;
$PickedBox \leftarrow []$;
; /* keep track of actions and box locations */
$bfsPaths \leftarrow BFS(boardState)$;
$selections \leftarrow bfsPaths.keys()$;
for *step* **in** *range(TrainingSize)* **do**
  Box positions ← Get all box positions;
  $Player \leftarrow Board.GetPlayerPosition()$;
  $currentState \leftarrow (Player, boxpositions)$;
  $Policy \leftarrow DeetermePolicy()$;
  if *Policy* **is** *greedy* **then**
    $selectedBox, action \leftarrow greedyPolicy(currentState, selections)$;
  else
    $selectedBox, action \leftarrow randomPolicy(currentState, selections)$;
  end
  Execute action on selected box;
  Update player location and box positions;
  $nextState \leftarrow$ (new player location, box positions);
  Update BFS paths;
  Update selections list with new BFS paths;
  $reward, BoxesDone \leftarrow$
   $Reward(state, BoxesDone, PickedBox, selectedBox, action, selections)$;
  Update PickedBoxActionList with (selected box, action);
  Update Q values(current state, selected box, action, reward, next state);
end

The algorithm runs faster than the MCTS since our state value table and the Q-value table is dictionaries. Given an action, it takes O(1) time to return the respective state value or Q value. Since we do not save board states in each state/episode, the space used is also reduced by a significant amount resulting in more iterations in the same amount of time. The Breadth-first search algorithm is the only algorithm that runs O(n**V) for the number of boxes and V nodes generated per box for finding a path. The main Q-learning algorithm has:

Time Complexity of n^m (n is number of possible box moves, m is the number of simulate steps)
Space Complexity n(n is the number of possible box moves)

3.2 Breadth First Search

The BFS pathfinding algorithm (Algorithm 3) is just a modification of the average breadth-first search. Hence the modification helps us search all possible paths to a given box or all box locations. By saving final paths once the location is reached, the BFS search continues for the remaining paths in the Queue and searches all possible options. The multiple paths mean the player can take either path to the box and move it in that direction. The choice of path depends on the player, and another modification can be added that checks if the path leads the current box into a deadlock and considers that path invalid. The neighbors of a position are the valid floor spaces that are not walls or other boxes. Instead of finding the shortest path, the algorithm finds all paths through valid floor spaces to a box. Certain boxes can have two

or more paths depending on the direction the agent approaches the box.

Algorithm 2: Breadth First Search Pathfinding

```
Input  : A box location coordinate
Output: A List of all paths to the box location
finalPaths ← [];
StartPosition ← board.GetPlayerPosition();
Queue ← [[Startposition]];
while Queue not empty do
    path ← queue.pop();
    position ← path[−1] ;                    /* Get last position in path */
    if position is box location then
        finalPaths.append(path);
        continue
    end
    for neighbor in getNeighbors(position) do
        if neighbor not in path then
            newPath ← path + neighbor;
            Queue.append(newPath);
        end
    end
end
return finalPaths
```

The time complexity of the proposed BFS algorithm O(n**V) where n is the number of boxes and V is the number of nodes generated for each box while looking for an optimal path.

3.3 Rational Action Simulator

This algorithm represents a simulation function that takes the current map, an action, and a step k to simulate the possible chance of getting deadlock within the next step k if taking the given action. It works similarly to the Monte Carlo search algorithm. It returns a count, representing our positive confidence of not being stuck into the deadlock situation after taking this action within the next k steps. Understanding that not getting stuck in time k is more important than not getting stuck in the time k + 1, so similar to the gamma value in reinforcement learning, we defined a variable in our function as a discount on the number of remaining spots to push in the future.

Algorithm 3: Rational Action Simulator

```
Input  : BoardState, Action, Step
Output: NumberOfPossibleMoves
newBoardState ← BoardState;
BoardState.movePlayer(Action);
NumberOfPossibleMoves ← 0;
for Action in BoardState.BFS() do
    NumberOfPossibleMoves ←
     NumberOfPossibleMoves + Simulator(newBoardState, Action, Step − 1);
end
return NumberOfPossibleMoves
```

3.4 Heuristics

3.4.1 Policy selection

First, we check whether the state passed is in the Q table or not. If the state is not present in the Q-table, the agent would either take random action policy or results from the rational action simulator function to explore the world. Otherwise, it chooses the greedy policy with some probabilities. The probability is controlled by both random() function and epsilon. The epsilon value drops as the number of training times increases.

3.4.2 Reward assignment

The reward function is used to calculate the reward after the agent pushes the selected box. When any boxes get pushed, it checks whether this is a new pushed-in box by comparing the board.boxesDone() with BoxesDone (to prevent a box from getting pushed in and pushed out repetitively). If a new box gets pushed in, the reward is 100. If the boxes have been pushed backward and forward, then the penalty is -5 times the number of times this box gets pushed (to avoid wasting time). If all boxes are in storage, the reward is 1000. If the box was pushed, but nothing happened, the reward is -5.

Algorithm 4: Reward

```
Input  : State, BoxesDone, PickedBox, selectedBox, action, selections
Output: Reward, BoxesDone
if any box reached storage then
    if board.boxesDone > BoxesDone then
        reward ← 100;
        Update number of total boxes done;
    else
        if PickedBoxActionList.count(selected box, action) > 2 then
            reward ← -5 for each action in the action list;
        else
            reward ← −5;
        end
    end
end
else if All boxes are in storage then
    reward ← 1000;
else if PickedBoxActionList.count(selected box, action) > 2 then
    reward ← -5 for each action in the action list;
else
    reward ← −5;
return Reward, BoxesDone
```

3.4.3 Q table update

The Q-table for the agent's coordinate is updated before pushing the box. The state of the Q-table is (agent's coordinate, boxes positions). Q-Predict represents the Q-value of (the agent's coordinate before the move, the boxes' positions). Q-target represents the Q-value of the agent's coordinate after the move. Then the Q-value for the old state is updated with the formula of

$$QTable[oldState][(selectedBox, action)] += \alpha * (Q_{target} - Q_{predict}) \quad (1)$$

where α is the learning rate.

4 Results

We have selected a few board problems from around ten inputs to include in this report. The success/failure of our agent on these boards will be discussed, along with what can be improved to solve this and why certain boards do not work as expected.

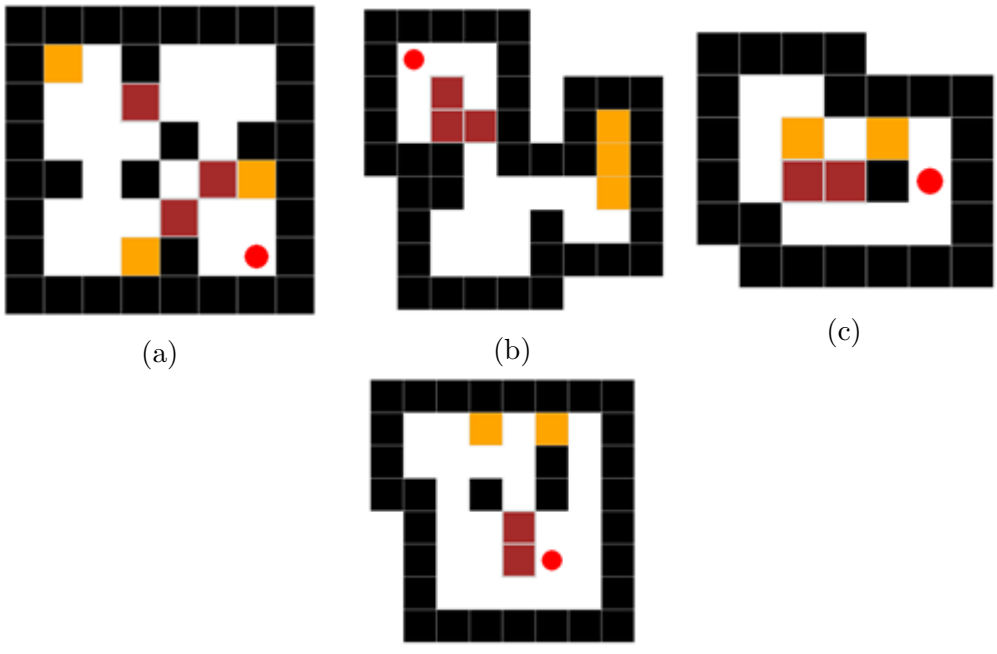
4.1 Successful experiments

Under a constrained training time, the Q-learning agent successfully manages to find a solution for boards that need multiple boxes to be moved to multiple storage blocks. The Q-table and state value table play a major role in this as once a box is moved to storage, the moves that lead to a good reward in the form of the box reaching the storage are preferred in the final run.

When the boxes are close to each other, our approach works as the number of reachable boxes is smaller. Our rational simulator can ensure that the agent has less chance of getting stuck into the deadlocks within a reasonable amount of time. Our approach's strength is that the rational simulator can guide the agent to make the most reasonable decisions as it helps the agent see several steps in the future.

The agent can pick which box to move first based on the initial positions using the Breadth-First Search multi-path finder, considering the box positions that may lead to an immediate deadlock, such as in the board (b) Figure 1. The initial training runs result in the agent learning from trial and error, moving boxes into a deadlock situation, and resetting the board, which gives a negative reward and is considered an undesirable outcome.

One change we made from our draft design is the rational simulator. It is designed to help the agent solve the Figure 1b. The tunnel in figure 1b makes our agent quickly stuck into the deadlocks. Thus, we added this simulator to make the agent make moves that cause fewer deadlocks in the future, which eventually helped us solve figure1b.



(a) (b) (c)

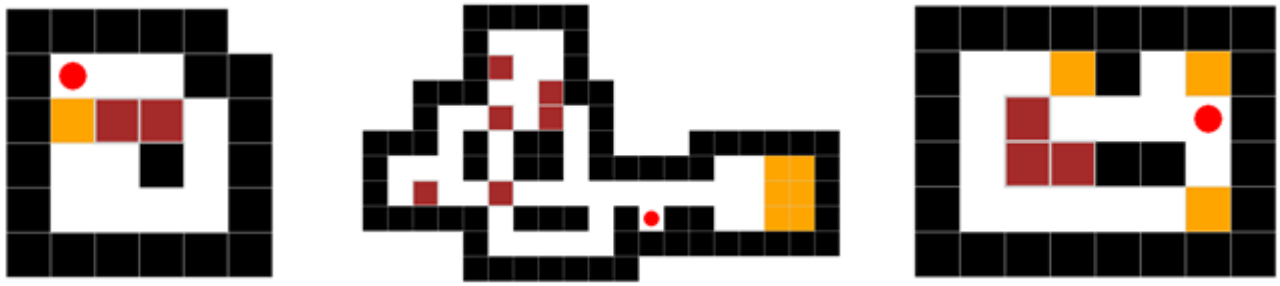
(d)

Figure 1: Working examples

4.2 Unsuccessful experiments

The agent, as it stands, is unable to find its way past certain situations where the decision-making is more complicated, and each move increases the number of possibilities and deadlock scenarios on the board.

Board problems such as (b) in Figure 2 present a logistical problem and a time-consuming problem for our agent since it has six boxes placed in a pattern where moving the wrong box first eventually leads to a difficult unsolvable deadlock to predict in advance for the agent. Since the board resets every failed run and a board state is not saved, the negative reward piles up on certain moves that may be a good option in certain situations down the line. The reset takes up a lot of time, and eventually, the problem is too time-consuming to solve for the agent under the given training time. Due to the way the agent detects deadlocks, moving a box right next to a box or a wall surrounding the box on two sides is not considered a great move. Moreover, in Figure 2b, our rational simulator would suggest the agent push the box with more possible reachable boxes within the next k-1 steps, but this is not true in this map. The boxes on the bottom straight lane path may get done easily compared to other boxes, but these boxes on the straight, narrow path would always have less reachable boxes in the future than other boxes. Therefore, our agent would never try to finish them first.



(a) (b) (c)

Figure 2: Unsuccessful examples

In cases of boards such as (a) in Figure 2, this is a situation that our agent cannot solve since it involves a box that is already placed on storage. The agent decides not to move the box out of storage as it is not an outcome that would be suitable and the goal of the algorithm as it is set up currently is to get the boxes to the storage locations. However, an enhanced heuristic could be introduced that would check if moving the box from the storage location would result in a path for another box to be moved to storage but generalizing this heuristic for larger boards was proving to be a difficult task.

In the case of boards such as (c) in Figure 2, our agent can successfully push two boxes into the storage. However, our agent cannot always push the third one correctly because the two storage that first gets fulfilled are the top middle one and the right corner one. Putting the right corner one would block the agent's way in accomplishing the third storage. We tried to use Manhattan Distance to guide the selection of boxes to push, but it does not work on this map as the two storage is at two corners so that the distance from the box to both of them are the same, which makes the Manhattan distance approaches nonfunctional.

The approach's weaknesses are the cost and the representation of the good move. The cost is the main issue that makes the agent slow on large maps because it needs to simulate all possible moves' futures. We represent the most rational move as if it has more possible reachable points in the future. However, this might be too aggressive. Sometimes, some boxes can only have one possible move toward the storage, while others have more. In practice, the box with only one move can sometimes be a better choice or sometimes not. Our approach is

too aggressive in always picking the one with more possibilities.

Considering the failure reasons we listed above, some future improvement suggestions we would make are thinking of some better heuristics in picking the boxes to push and getting better representations for our rational simulator to solve maps like Figure 2b.