Sokoban Final Report

Emily Brunelli, 23805145 **Ricky Cheng, 81593172** **Andreana Chua, 52488016**

1 Intro

Sokoban is a puzzle game that was developed by Hiroyuki Imabayashi in 1982. The game takes place in a warehouse, with some number of boxes along with an equal number of storage spots. The objective of the game is for the player to push all the boxes onto storage spots.

Although the game rules are simple, the problem of solving Sokoban is not at all simple. In fact, Sokoban is shown to be NP-hard [2] and PSPACE-complete [1]. The game is also difficult due to its large branching factor, exponentially deep search tree, and the existence of deadlocks.

Thus the problem of Sokoban is interesting since it provides a simple model to the real-life application of AI in the form of autonomous warehouse robots. The irreversible nature of deadlocks means that agents must either learned some policy for actions or perform some form of planning ahead of time. Human solvers can easily spot features within the puzzle as they plan their moves. Machines, however, require vastly more computations to solve a level. Some solvers, such as the Rolling Stone [4], utilize a single agent with IDA* search combined with various feature extraction techniques to solve the problem. In our project, the agent will be using Q-learning (using ϵ-greedy to explore) with a BFS subroutine and deadlock elimination to play Sokoban.

2 Algorithm Setup and Preprocessing

When reading in the input, we utilize the same symbols that are presented in the project manual, including a symbol of our own, which will be described in the next section. This includes:

Sokoban Object	Symbol
WALL	'#'
FLOOR	blank
BOX	'$'
GOAL	'.'
AGENT	@
DEADLOCK	'x'

We introduce the negative reward for FLOOR and assign all FLOOR tiles to be -1. WALL tiles are assigned a constant that is checked when the agent considers a move, such that it is impossible for the agent or the box to be moved inside the wall. When the agent places a box on a GOAL space, we give a reward of 1000.

Ultimately, in order to push any given box, the agent will need to move to a square directly adjacent to it. For this reason, a dictionary will be stored that includes every box and a list of paths the agent may take to reach any square adjacent to the box (and vice versa, a list of paths from box to goal). It won't be necessary to use an algorithm that considers weights here (such as Dijkstra's algorithm or A*), and for this reason, we can simply use BFS. First, we determine which locations a box can be legally moved into (i.e. do not contain other boxes, are walls, or are non-goal deadlocks), and then for all boxes, run BFS to find the path required for the agent to move from its current position to the position required to push the box. After storing a dictionary of all the valid paths, we run epsilon-greedy to either randomly sample from the paths for the agent to take, or we choose the path that results in the highest q-value from a box being moved.

It is possible for the BFS path-finding algorithm to not find any valid paths. In this case, we give the agent a negative reward of -10000, discouraging it from returning to that specific state again.

As mentioned earlier, we add deadlock positions on the board, in order to prevent the agent from pushing the box on these positions. This is described below.

3 Deadlocks

One way that we have decreased the size of the state space is by removing the capability for boxes to be pushed into some forms deadlock positions. A deadlock occurs when a solution is no longer reachable, since it is impossible to move all the boxes in the goal states. This can occur when a box has entered a position in which it can no longer be moved to a goal state, such as corners. Notably, while a box should always be prevented from reaching a deadlock that is not in the same "deadlock zone" as a goal, the agent has no such restriction. There are, however, several situations in which the agent will be forced to push a box into a deadlock; for example, in the toy 5x3 map example, there is a hallway of width 1 that is technically a deadlock zone. A check will be put in place to verify if a box must enter a deadlock. There are two types of deadlock positions that we found from [3]. One is called static deadlocks and the other is called indirect deadlocks.

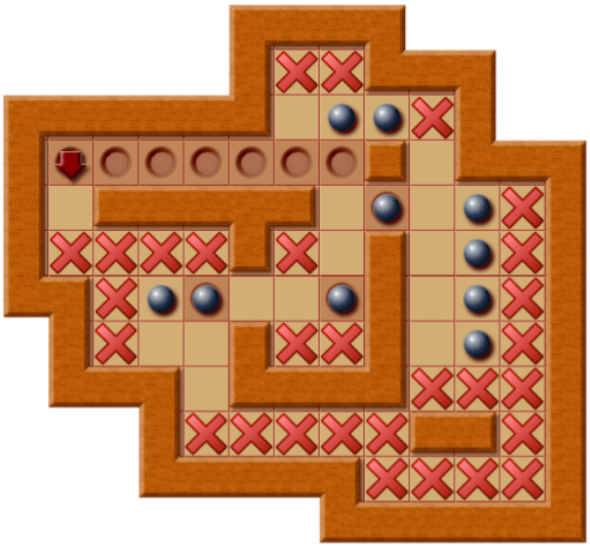
Static deadspots are another way to say a box has been moved into a corner. Once in a corner, a box can no longer be pushed by the agent. A position is formally a static deadlock when:

1. The position is not a goal position.
2. There are at least 2 walls next to the position.
3. At least 2 of the walls next to the position are not on opposite sides of that position .

The other type of deadlock is when a box moves into an indirect deadlock. These positions would indirectly cause a deadlock because a player cannot immediately determine that these positions would cause a deadlock. For example, if there are no goals against a wall and the agent were to push a box against a wall, it would result in a deadlock since the box cannot be pushed away from the wall and is thus stuck against the edge of the board game. This would result in the game not being solvable. Formally, a position would be an indirect deadlock when:

1. The position is not a goal position.
2. The position is not a static deadlock.
3. A box placed on the position cannot reach another position that is not a static or indirect deadlock.

Below is an image of a Sokoban board with each deadlock indicated by an "x", with all corners being static deadlocks and all other deadlocks being indirect deadlocks.



With these ideas in mind, we determine if a position is a deadlock by executing Algorithm 1.

Algorithm 1 Deadlock position finder

```
    Input: Sokoban board.
    Output: Sokoban board with deadlock positions marked.
 1: procedure DEADLOCK(board)
 2:     Mark all non-walls as deadlocks
 3:     Mark all non-walls as not yet processed
 4:     Mark all goals as non-deadlocks
 5:     for every position which is not yet processed and non-deadlocks do
 6:         Pos1 = this position
 7:         for each direction do
 8:             Neigh1 = position in direction
 9:             if Neigh1 is not processed and position from Neigh1 could be pushed to Pos1 then
10:                 Mark Neigh1 as non-deadlock
11:             end if
12:         end for
13:         Mark Pos1 as processed
14:     end for
15: end procedure
```

Along with decreasing the size of the state space, deadlocks are also used to train the agent. If an agent pushes a box onto a deadlock position, we give a negative reward of -1000. Because the episode is terminated once a deadlock position is reached, we don't give the agent a chance to potentially gain rewards from the other boxes. If it could, then it would incorrectly learn that leaving a box in a deadlock position could potentially be a good state to be in. Given that and the high negative rewards, it discourages the agent from pushing the boxes onto these deadlock positions.

4 Q-learning

After creating the board, we utilize Q-learning to train an agent to play Sokoban. Q-learning is a Reinforcement Learning technique that is used to find the optimal policy in a Markov Decision Process. When the optimal policy is found, the agent is able to make the next best action at a given, current state.

To find this optimal policy, the algorithm iteratively updates Q-values for each state-action pair as the agent traverses through the environment. Eventually, these Q-values will converge to the optimal Q-values for each state-action pair, resulting in an optimal Q-function, which is what Q-learning is based off of. The Q-function is a function of a state, s, and an action, a, for a given policy and it calculates the expected return for taking action a in s and following the given policy. The optimal Q-function is defined below:

$$q_*(s, a) = max_\pi q_\pi(s, a)$$

Along with the optimal Q-function calculating the largest expected return an agent can receive from a policy, it must satisfy the Bellman Optimality Equation, which is defined below:

$$q_*(s, a) = \mathbb{E}[R_{t+1} + \gamma max_{a'} q_*(s', a')]$$

This equation ensures that for any state-action pair at time t, the expected return for when the agent starts in s, selects action a, and follows the optimal policy thereafter is going to equal the expected reward the agent gets for taking action a in s, which is R_{t+1} plus the maximum expected discounted return that the agent can obtain from the next state-action pair.

When an agent picks an action at a state, it will observe some reward after taking that action. When it does, it'll be able to update the Q-value associated with that state-action pair. It will update the Q-value with the following equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma max_a[Q(s',a')] - Q(s,a)]$$

In order to keep track of all this information, a Q-table is created, in which each row represents a state the agent is in, and each column represents the possible actions an agent can take in a state. In this case, there will be $h \times w$ rows, one for each position in the board game, and 4 columns, one for each of the four directions. We also added another dimension to keep track of box positions. So overall, each state will be the agent's and board's state. The Q values will be initialized to 0 at the very start.

Since the Q-table has virtually no informative values that the agent can use, we utilize the epsilon-greedy algorithm, so the agent can either explore the game board, or exploit the current values in the Q-table. We chose $\epsilon = 0.8$, meaning there is a .2 chance of random exploration.

Given all of our components, we execute Q-learning as described in Algorithm 2, while also scaling the rewards by a quadratic function of step such that faster solutions are preferred over slow ones.

Algorithm 2 Q-learning

Input: `None`
Output: `Updated, optimal q-table`
`Initialize Q-table as described above`

procedure Q-LEARNING
  for `each episode` **do**
    `Initialize s`
    for `each step of the episode` **do**
      if random-num > epsilon **then** ▷ Perform epsilon-greedy algorithm
        `Choose agent path a via exploitation`
      else
        `Choose agent path a via exploration`
      end if
      `Take agent path a`
      `Observe reward r and next state s'`
      $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma max_a[Q(s',a')] - Q(s,a)]$
      $s \leftarrow s'$
      if s is terminal **then**
        `break`
      end if
    end for
  end for
end procedure

Algorithm 3 illustrates the algorithm we used to find paths that may be taken by the agent.

Algorithm 3 BFS pathfinder

Input: `Sokoban board, start position, end position, current box positions.`
Output: `A path from the start position to the end position.`

```
procedure PATHFINDER(board, start, end, currBoxes)
    Make copy of board, replacing all positions that are accessible with 0's and all inaccessible positions with 1's.
    Create matrix with same size of board initialized to all 0's, place a 1 in the position corresponding to start.
    Set step = 1
    while End position at matrix is 0 do
        Expand from current position to all valid non-wall and non-box locations
        Update matrix in expanded positions to step + 1
    end while
    Set current matrix position to end
    Set path = []
    while Not at start position do
        Find neighbor of current position equal to step - 1, add coordinate to path, and then decrease step by 1
    end while
    return path
end procedure
```

5 Complexity Analysis

Solving for Sokoban is highly dependent on the number of boxes the level may have. Solving for $k + 1$ boxes yields a different set of solution path than solving for k boxes.

The space complexity for the Q-table is thus also correlated with the number of boxes there are. Our Q-table is represented by the coordinate of the boxes and the boxes may be in any of the floor squares. Therefore, for a level with n floor squares and k boxes, the space complexity of our algorithm is $O(\binom{n}{k})$, or $\Theta(n^k)$. This discovery is in contrast to our earlier estimate in the draft, where we gave a space complexity of $O(kn)$.

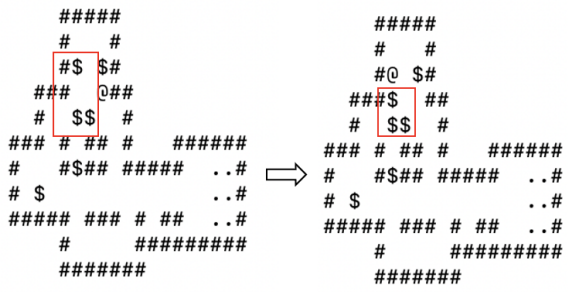
For time complexity, Q-learning is on order $O(mn)$ where m is the number of Q values and n is the number of states. BFS is $O(b^d)$ where b is the branching factor (being 4, because there are 4 directions) and d is the depth, and so we expect the time complexity of BFS and Q-learning combined to be to be $O(b^d mn)$.

6 Experimental Results

In order to demonstrate our agent's performance, we used the benchmark problems as part of our experiments. In total, the agent was able to solve the first six input files in a few milliseconds. The results are shown below:

Test	Time (s)
sokoban-01.txt	0.018
sokoban-02.txt	0.065
sokoban-03.txt	0.045
sokoban-04.txt	0.406
sokoban-05a.txt	0.093
sokoban-05b.txt	0.237

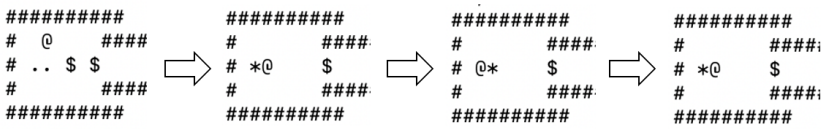
Once the board and the number of boxes increased in size, our agent failed to solve the game. Observing its movements for some of the episodes, it tends to push boxes into deadlock positions, mainly ones that involve pushing two boxes together, next to a wall, making it impossible to move them. For example, in $sokoban01.txt$:, our agent pushed the top left box down, causing a deadlock situation:



With a board with narrow hallways and multiple boxes, our agent had to experience multiple deadlock positions, which would cost an episode of learning. Whenever a box moved, it would introduce new deadlock situations, which increased the number of states that the agent would have to explore. Considering this, a weakness of our approach was the large state space. Since we consider the agent's position and the boxes' configuration as one possible state, there are a large number of states the agent will have to encounter, as seen in the benchmark problems starting at $sokoban01.txt$. As a result, it would continuously end up in deadlock states, reaching the maximum number of episodes we set before being able to learn where the goal states were. Because of the nature of the board, the agent would have to experience multiple deadlocks before making any true progress. This progress could not be achieved within the one hour time limit.

The reason why our agent could solve the first six problems was because the board was relatively small, there was a few amount of boxes, and most importantly, there was room to move the boxes around, which prevented the introduction of new deadlock positions. With our BFS path-finder algorithm, it was able to find a clear path from box to goal, without being blocked by other boxes. If the agent were to push a box into a deadlock position, it would quickly learn that the deadlock state is something to avoid, since the state space isn't quite that large.

Another weakness we observed was that if two goals are next to each other, and the agent pushed a box to one of the goals, in subsequent moves, it would continuously move the box between the two goals, as shown below, on a test board, in which $*$ means box on goal:



This made sense, since BFS still considered the box as a possible object the agent could move, and the closest goal that wasn't under the box, is the one next to it. Compared to the other box that is much farther from the set of goals, it is more ideal for the agent to push the box to the other goal and gain the reward. With this continuous back and forth, the agent is able to achieve maximum rewards, but not make progress to the overall goal.

Though, in terms of finding closest goals, we observed that BFS was a good choice for finding paths from boxes to goals, when the conditions were ideal. Every move that the agent decides to make would make the box closer to the goal, which helped speed up the agent's learning, since it wasn't making random moves at every step. Although there could've been some improvements to the algorithm, using BFS allowed the agent to successfully push boxes to goals in an efficient way.

7 Discussion and Conclusion

As mentioned above, one shortcoming of our current AI is that boxes would get into deadlock positions involving other boxes. One immediate improvement that could be made is to factor other boxes into the deadlock checks. There could also be mechanisms put in place to prevent boxes from moving back on goal locations for infinite rewards; while we don't want to permanently lock a box into a spot just because it has found a goal, we may want to change the reward scheme such that it's no longer incentivized to move away unless required by another box.

There were several optimizations we could have made regarding our BFS subroutine. As is, at every step, we run BFS a number of times equal to the number of valid positions a box can be moved into (and then storing a dictionary of possibles boxes to move and the corresponding paths that must be taken by the agent). This is fairly wasteful, but was initially done because of complexities that arose when attempting to randomly sample actions when epsilon greedy triggers. An improved version of our BFS strategy could involve first checking the q-table for the action with the highest q-value, and verifying that it is reachable by the agent using BFS and returning the corresponding path. If it isn't reachable, the next best q-value is used and BFS performed again, etc. When epsilon-greedy triggers, we randomly sample an action, verify using BFS that it is reachable by the agent, and move the agent using the corresponding path if it is reachable. If it isn't reachable, we sample without replacement to re-choose an action, and continue this until finding a new valid action.

During our development, we made the decision to shift from a q-table that was of size BoardSize·Actions, where Actions = 4 (because a box may be moved up, down, left, or right) to a q-table that was NumBoxes·BoardSize·Actions. As a result, drastic improvements were seen on some levels, while it performed slower on a minority of others.

A link to our repo: https://github.com/rickysrcheng/CS271Sokoban

References

[1] Joseph Culberson. Sokoban is pspace-complete. Department of Computing Science, The University of Alberta, 1997.

[2] Dorit Dor and Uri Zwick. Sokoban and other motion planning problems. pages 215–228. Computational Geometry Volume 13 Issue 4, 1999.

[3] Nils Froleyks. Using an algorithm portfolio to solve sokoban. pages 29–31. Department of Informatics Karlsruhe Institute of Technology, 2016.

[4] Andreas Junghanns and Jonathan Schaeffer. Sokoban: improving the search with relevance cuts. pages 151–175. Theoretical Computer Science 252, 2001.