A Q-Learning Approach to Sokoban

Jonathan Ta **Jhih-Yuan Chang** **Elia Rho**
54083468 87921851 63058004
tajk@uci.edu jhihyuac@uci.edu jsrho@uci.edu

December 18, 2020
Donald Bren School of Information and Computer Science

1 Introduction

Sokoban is a puzzle game where a single player pushes boxes to different storage locations in a warehouse map. Each map contains a list of movable boxes, immovable walls, a list of storage locations, and the player's initial starting position. Players cannot move into immobile objects such as walls, but can push boxes given that the box has an empty location to be pushed to. To complete the game, the player must push a box onto each storage location marked on the map. Certain maps may have more boxes than marked storage locations.

The Sokoban game is known to be difficult to solve despite its simple rules considering its incompatibility with heuristics and its large state space. In particular, Sokoban is known to be NP-hard [2]. Each movement of a player or boxes yields unique game states, leading to a large state space even for small maps. For example, a 8×8 map may consist of $7^2 = 49$ accessible tiles and a boundary wall along the sides of the map. For this map, 3 boxes would yield a total of $\binom{49}{3+1} = 211,876$ unique arrangements of boxes and player.

Furthermore, not all map configurations are solvable. For example, in maps with equal amounts of storage locations and boxes, any frozen box configuration (e.g., any box with directly adjacent walls) makes the map unsolvable. These situations, known as *deadlocks*, require proper identification and avoidance for efficient exploration of the state space. In fact, a major difficult in Sokoban maps stems from the deadlock navigation instead of getting boxes to the goal.

Previous approaches to solving Sokoban used a variety of approaches, including informed search and reinforcement learning. Informed search uses a foundation of Iterative Deepening A* Search (IDA*) in combination with domain-specific knowledge. These improvements (e.g., deadlock tables or move ordering) implement theoretical concepts such as memorization and search scheduling. Although optimal and intuitive, these search methods tend to be semi-exhaustive and require a lot of time. Reinforcement Learning (RL) approaches utilize Markov Decision Processes (MDPs) and state-action functions to develop policies. In addition to benefiting from the domain-specific techniques above, RL approaches have the added advantage of avoiding the exhaustive behavior that search algorithms have.

Given the large search space, we use Q-Learning as the basis of our approach. We first develop a standard Q-agent with the most intuitive state space, then present two designs that build upon the traditional Q-learning method. The Q-learning with a non-traditional state space and the deep Q-learning implementations reside in the `QAgent, BoxAgent, DeepQAgent` classes in the source code. In the following sections, we introduce the basic data structures necessary for each algorithm in section 2, the three algorithm variations in section 3, observations and analysis in section 4, and the conclusion in section 5. In the appendix, we provide records of the runtime as benchmarks.

2 Data Structures

2.1 State

Given a Sokoban map, any configuration of the map can be uniquely represented by the player's location, and the locations of the boxes. For every moves of the player, state is updated and check whether the state is a goal state. The goal state is any state where k boxes have equivalent positions to k storage locations. Until the state becomes the goal state or the box is at deadlock position, state is updated.

In our approaches, we use two distinct implementations of State. For standard Q-Learning, we encapsulate state in an object along with a two-dimensional array of the current map, a list of boxes, a set of storage spaces, and the current location of the player:

```
class State:
    map = [[0, 0, 0, 1, 0, ...], ...]
    boxes = [[x, y], ...]
```

```
    storage = set([[x, y], ...])
    player = [x, y]
```

In contrast, Deep Q-Learning utilizes an alternative state implementation derived from previous approaches using an one-hot encoded multi-dimensional array of each type of map element, i.e., wall, box, player, and storage. The first index represents the type of map element, and the remaining two dimensions describe the one-hot encoding for the existence of an object. For example, a box at location $(3, 4)$ would be stored in `state[1, 3, 4]`.

```
state = np.zeros((4, xlim+1, ylim+1))
#four planes for states of wall, box, player, and storage
```

2.2 Action

A player can choose to move in the four orthogonal directions on the xy plane: up, down, left, right.

```
UP = np.array([0,1])
DOWN = np.array([0, -1])
LEFT = np.array([-1, 0])
RIGHT = np.array([1, 0])
DIRECTIONS = [UP, RIGHT, DOWN, LEFT]
```

2.3 Q table

The Q-Learning algorithm requires a table with all Q values for some state and actions.

```
q_table = {(state, action) : q_value, ...}
```

2.4 Path table

For our alternative state space, path finding is an important aspect. To avoid incurring redundant performance costs, we store previously calculated paths for a given state in a path table:

```
path_table = {(state, destination) : [action1, action2, ...], ...}
```

We use the state and destination as a key since the origin of the path is the position of player in a state.

2.5 Deadlock table

Many approaches make use of a deadlock table to avoid repeated computations on previously seen states. We implement a similar design in our environment. Our deadlock table uses a python dictionary to store a list of index-specific box states per hashed state.

```
deadlock_table = {(state) : [box1_frozen, box2_frozen, ...], ...)}
```

2.6 Environment

The environment contains all information regarding the Sokoban map. This includes not only a internal logical structure of the map, but also a list of positions for every boxes and storages.

```
class Environment():
    state = ...
    deadlock_table = {state : [...]}
```

2.7 Agent

The agent manages all information and logic regarding the algorithm processes. This includes the Q-table, any cached 'experiences', or any algorithm parameters. This becomes the superclass data structure for specific agents such as `QAgent`, `BoxAgent`, or `DeepQAgent`.

```
class Agent():
    qtable = {(state, action) : q_value, ...}
    experience_cache = [[action1, action2, ...], [action1, action2, ...], ...]
    learning_rate = 1 #example
    discount_rate = 0.8 #example
```

3 Algorithms

In our work, we show three different variants of Q-Learning approaches to solve the Sokoban problem. Each algorithm has distinct advantages and disadvantages.

3.1 Q-Learning

Q-Learning is a reinforcement learning technique that utilizes the following Bellman Equation to improve the policy:

$$Q'(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(R(s_t, a_t) + \gamma \cdot \arg\max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)).$$

Here, for each time step t, s_t is the current state, a_t is the chosen action at the given step, $\alpha \in [0, 1]$ is the learning rate, $\gamma \in [0, 1]$ is the discount factor, s_{t+1} is the next state, and $R(s_t, a_t)$ describes the reward function. For the particular case of Sokoban, it is ideal to use a learning rate of 1 since the game is deterministic. Below, we present the general algorithm to update the Q-table using Q-Learning across different episodes.

Algorithm 1: Q-Learning

Require: s_i, α, γ
  $\forall s \forall a\ Q(s, a) \leftarrow 0$
  while not converge() **do**
    $s \leftarrow s_i$
    while not goal() **and not** deadlock() **do**
      a $\leftarrow$ choose_action(s)
      for all a' **in** actions **do**
        if max **not** intialized **or** max $<$ Q(s', a') **then**
          max $\leftarrow Q(s', a')$
        end if
      end for
      $Q(s, a) \leftarrow Q(s, a) + \alpha(R(s, a) + \gamma\text{max} - Q(s, a))$
    end while
  end while

Algorithm 1 represents a sample episode for the Sokoban game. At each step in the episode, the Q-learning algorithm chooses an action, then updates the Q-table value for the current state and action. Choosing an action is dependent on whether the agent is exploring (i.e., using the epsilon-greedy metric) or the agent is evaluating the policy (i.e., choosing the $Q(s, a)$ that yields the highest value).

Once we have found a policy that yields a solution and or deemed sufficiently optimal, we can utilize this policy to solve our problem. In our implementation, we stop the moment the algorithm evaluates

the policy to find a goal, which results in a non-optimal solution. In particular, we present the policy evaluation algorithm in Algorithm 2

Algorithm 2: Policy evaluation

```
Require: s, Q(s, a)
  for all a ∈ A do
    if m not intialized or m < Q(s, a) then
      m ← Q(s, a)
      a' ← a
    end if
  end for
  return a'
```

3.1.1 Complexity

Each Sokoban map can have 1 player, b boxes and t total tiles. However, certain combinations of boxes are impossible to reach or complete – these configurations are known as deadlocks. To calculate an approximate for the total number of states, it suffices to calculate the upper bound, without considering deadlocks. Then, the number of states is equivalent to the number of possible combinations for boxes and player on the board, or $\binom{t}{b+1}$. This large state space then requires proper exploration and learning to efficiently solve a given map.

Time Complexity For each episode, we perform a sequence of actions. We can bound this sequence arbitrarily for our algorithm by resetting the state whenever we reach our maximum iteration count. Let the iteration bound be called l such that for a solution with length l_g follows $0 < l_g < l$. Then, we can view the system 'waiting' for an episode to produce a sequence of actions that results in a goal. Since this is analogous to the coupon collector's problem, we join this with our bound to form a time complexity of $O(l * slog(s))$ [4].

Speedup It is worthwhile mentioning that since we are effectively performing many iterations of random walks, there exists parallel speedups for covers and hit times of random walks with k independently parallel random walks [1]. Future work involving these sort of solvers would therefore benefit from a parallel algorithm that utilizes this characteristic.

Space Complexity The space complexity is bounded by our Q-table and deadlock table sizes which together is $s * a + s$ for the number of actions a. Therefore, the space complexity becomes $O(s * a)$.

Overall, the time complexity of the Q-learning algorithm can be quite costly solely due to the large state space of the Sokoban state space. To amend the costs of the large state space, an efficient exploration policy is needed to reduce unnecessary exploration.

3.1.2 Exploration Policy

The Q-Learning algorithm improves the policy via exploring the map. A naive policy may use a completely random exploration policy to choose actions. However, as demonstrated above, this can lead to the worst-case time complexity with an abhorrent time cost. To avoid this, we adapt some techniques that will improve time complexity while keeping the policy random for diverse exploration of the state space.

Epsilon-greedy exploration As we improve our policy, it becomes beneficial for the algorithm to explore along the expected goal path for different routes to the goal. For some ϵ, we have a probability ϵ for the algorithm of using the policy to choose an action instead of random exploration. In particular, we linearly scale ϵ with respect to the number of episodes run. ϵ saturates at 0.8, enabling greedy properties yet still allowing for exploration.

Back propagation The Sokoban state space is characterized with sparse rewards in comparison to actions taken. In plain English, this means that the agent spends a lot more time exploring open space, and only a small amount of time receiving rewards. This means that the possibility of reaching the goal and receiving rewards, but not updating the state-action function $Q(s, a)$, is high. To amend this, we replay the action sequence in reverse, effectively propagating the newly updated goal state-action value $Q(s_g, a) = R(s_g, a)$ to other parts of $Q(s, a)$.

Action Pruning Often, there are actions that are guaranteed to be invalid or never beneficial, such as actions leading to deadlock states, or actions that are 1-length cycles. To avoid exploring these states, we prune these actions so that the agent does not consider them.

3.1.3 Reward function

In Sokoban, the goal state is reached by putting each box on storage. To this end, we will give the reward every time the agent pushes the box to a storage, and gives a negative reward when boxes are pushed off. This behavior is to prevent feedback loops, where the agent continuously pushes a box on and off a goal to generate infinite reward.

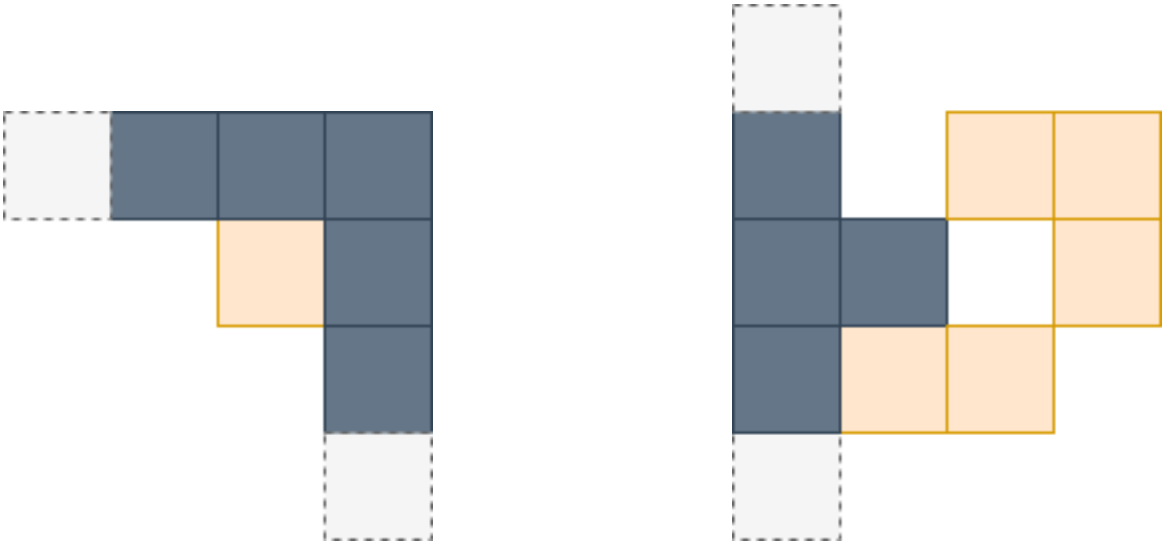
```
def reward(state, action):
    if goal_reached:
        return 500
    elif count_goals(state) < count_goals(next_state):
        return 50
    elif count_goals(state) > count_goals(next_state):
        return -50
    elif is_deadlock(state):
        return -2
    else:
        return -1
```

3.1.4 Deadlock detection

Deadlocks refer to states where the Sokoban game cannot be solved. One of the most common deadlocks is the scenario of a box in the corner of a wall intersection, where the box is frozen since there exists no method of moving the box. Since deadlock states are undesirable from both an exploration and search perspective, detecting and avoiding deadlocks is crucial to achieving good performance. In the case of our Q-Learning algorithm, we terminate immediately upon detecting a deadlock so that we do not waste time exploring the states within a deadlock.

There exists a large variety of deadlocks, and many of them are difficult to detect. As mentioned above, the corner deadlock is a classic example. This type of deadlock can be extended all frozen-type deadlocks, where a box is frozen due to other surrounding boxes or walls. Another one is the dead diagonal deadlock, where the center of a group of boxes is unreachable, causing several boxes to be frozen. Figure 1 shows an example of both types of deadlocks.

Below, we provide a general description of the algorithm necessary to identify these deadlocks.



(a) Corner deadlock. (b) Dead diagonal deadlock.

Figure 1: The two deadlocks in question.

```
isDeadlock(state):
    for each box in state.boxes:
        if box in deadlock_table:
            return True
        if isFrozen(box):
            return True
        if isDeadDiagonal(box):
            return True
    return False

isFrozen(box, previous):
    if box in previous:
        return True
    neighbors = getNeighbors(box)
    previous.add(box)
    for two adjacent neighbors:
        if both neighbors are walls or both neighbors are isFrozen(boxes, previous):
            return True
        else if one is wall and the other is isFrozen(box, previous):
            return True
    return False

isDeadDiagonal(box):
    for neighbor in neighbors(box):
        surrounding = neighbors(neighbor)
        unreachable = all(surrounding is wall or box)
        if unreachable:
            diagonal_pair = get_diagonals_pairs(center)

            if diagonal_pair is wall or box:
```

```
                return True
    return False
```

Alternative techniques Although we already prune the deadlocks directly through action pruning, sometimes the neighboring states are also deadlocks despite not having boxes frozen. A simple method of detecting deadlocks can be checking for repeated states in an episode. For example, if a specific state occurs more than 3 times in a short succession, we can consider the given state a deadlock and reset the state. Specifically, the likelihood of the same state occuring three times in a short succession for a length-2 cycle is $\frac{1}{|A|}^6$, where $|A|$ is the number of actions available. For a standard set of 4 actions, we determine that predictions are sufficiently improbable such that the benefits outweigh the costs.

Other deadlocks There exists even more varied deadlocks, such as corral deadlocks, a more general version of the dead triangle, or unreachable deadlocks, where the player is unable to reach certain locations. Corral deadlocks are a type of general deadlock where regions of the board are unreachable due to boxes, resulting in restricted movement for those boxes. In hindsight, corral deadlocks in particular would have been a candidate for implementation. However, these deadlocks are not only complicated but also computationally expensive to detect. In our solver, we choose the two simplest deadlocks, corner and dead diagonal deadlocks, to prioritize simplicity and speed.

3.1.5 Goal detection

The goal of Sokoban is moving each box onto storage. Therefore, we can simply check if all the boxes are at the storages.

```
isGoal(state, map):
    for box in state.boxes:
        if box not in map.storage
            return False
    return True
```

3.2 Q-Learning with a Non-traditional State Space

In the standard state space, there are four maximum possible actions at each step for the player: up, down, left, right. Although intuitive, this action scheme often results in a lot of repetitive exploration for simply pathing the player to the boxes. To amend this, we propose an alternative agent state space where the possible actions instead describe the possible movement of the boxes. For example, the possible actions for a map with a single box would be the four possible directions a box could move: up, down, left, right. Then in lieu standard player movement, an A* algorithm is used to path to position required to move the box.

In this state-action space, actions can no longer simply indicate a direction without specifying the box to act on; we redefine the new action data structure as a tuple specifying the box, and the direction it moves: `box_action = (box, action)`.

Immediately, the largest benefit for this state space is removing the need for Q-Learning to maneuver and explore hallways, where randomized exploration suffers. Instead, the Q-Learning algorithm focuses on box manipulation, leaving path finding around obstacles to a more well-established algorithm A*. This allows the algorithm to reduce the redundant cost of revisiting states near the starting state– a common problem for random walks.

3.2.1 Optimality

With the change in state space, it is not immediately apparent that the solution to the Q-Learning problem would be optimal given enough time. We consider two methods of solving for an optimal solution, **although our solver does not implement it**.

Move ordering Within Q-Learning itself, the agent is not aware of the distance between the player and the box in any state. As such, the agent will eventually find a solution with the least possible amount of box moves, but the order may be out of sequence, i.e., an derangement. It is then sufficient to place an order on the possible actions of the agent with respect to path cost.

Path smoothing Once the Q-Learning algorithm returns with a set of box-actions, we can transform this into a list of standard actions on the player with the help of the path table. However since each path was originally separate, it may be necessary to reorganize the paths when merging.

3.2.2 Complexity

The overall time and space complexity of the algorithm does not change. However, the number of states possible in this state space drops by a factor of $\frac{4b}{t}$, as number of unique player positions is effectively reduced only to positions adjacent to the boxes. As a result, the time complexities and space complexities drop a similar factor despite being equivalent asymptotically.

3.3 Deep Q-Learning

Deep Q-Learning is a promising branch of Q-Learning that utilizes function approximating characteristic of neural networks. In deep Q-learning, we use Q-function instead of a Q-table to calculate the state-action value with parameters $\boldsymbol{\theta}$. We minimize the error between the Q-function and optimal Q-function such that $Q(s, a; \boldsymbol{\theta}) \approx Q^*(s, a)$. For each step i, we measure the error by the loss function: $L_i(\theta_i) = \mathrm{E}_{s,a,r,s' \sim \rho(.)}[(y_i - Q(s, a; \theta))^2]$ where $y_i = r + \gamma Q(s', a'; \theta_{i-1})$.

Algorithm 3: Deep Q-Learning

Require: s_i, α, γ Initialize Q-function with random parameters θ
  while not converge() **do**
    while not goal() **and not** deadlock() **do**
      a $\leftarrow \max_a Q(s, a; \theta)$ Get reward r and s' by performing the action
      $s', reward \leftarrow s_i$
      $y \leftarrow r + \gamma max_{a'} Q(a', s; \theta)$
      Update θ with gradient descent with loss function.
    end while
  end while

Through the process of adapting deep Q-learning to the Sokoban problem, we designed two different implementations. One is training on the given map with only the positions of player and boxes as input and approximating the Q-function for this specific map. In the second one, we try to implement a general map solver whose network can output Q-values for any given map.

3.3.1 Further considerations

One-hot encoding One hot encodings are a standard method of representing features for neural networks. For unrelated features like boxes and players, one-hot encoding improves training efficiency by preventing the model from inferring non-existent relationships in integer encodings.

Given a map, the walls and goals are fixed, so we only need to know the position of boxes and player. To build a neural network, we firstly flatten the 2D map into 1D array and then apply one-hot encoding to *player* and *boxes* as the input. The network will output Q-values for each action.

Experience Replay Due to the sparsity of rewards in Sokoban, training the network immediately after the action is inefficient. To improve the efficiency, we introduce Experience Replay which provides two benefits: (1) having the network train on the data which includes rewarded actions, and (2) minimizing the correlation between the serial actions and corresponding states [3]. During the training episode, all the actions taken by the agent will be put into replay buffer along with the information of reward, previous state, and the state after the action, and for each step, we randomly choose n samples from the buffer to train the network.

Reward normalization Similar to the original work in deep Q-learning, we normalize the rewards given to the agent to improve training efficiency and prevent exploding gradients. Although this may hamper the model's understanding of different reward magnitudes, the Sokoban game allows uniform rewards given its simple rule set for counting score. Below, we provide a rough overview of the normalized reward function:

```
def reward(state, action)
    if goal_reached:
        return 1
    elif count_goals(state) < count_goals(next_state):
        return 1
    elif count_goals(state) > count_goals(next_state):
        return -1
    elif is_deadlock(state):
        return -1
    else:
        return 0
```

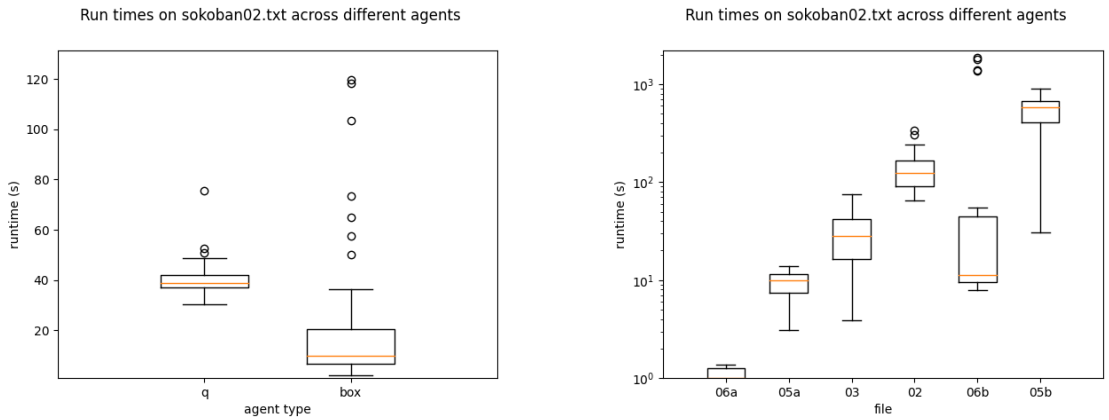
Figure 2: Normalized reward function.

3.3.2 Convolutional Neural Network

Since Deep Reinforcement Learning achieved human-level performance on Atari 2600 games [5], we modify this method to solve the Sokoban problem. We firstly encode the sokoban input into a 4-layer map – player, boxes, walls and goals. Then, we build a convolutional Neural Network which takes the encoded map as input to approximate Q-values for all actions in a state.

Using Mnih et. al's work as a basis, we construct a similar network architecture in hopes of extending their work to the Sokoban problem. First, the input of a map is padded to meet the network size. For example, for an 8×6 map, we would pad ends of the array with zeros such that the resulting one-hot image of the Sokoban map was 15×15 (specific to our implementation). We then fed this image through four convolution layers pair with batch normalization layers, followed by three fully connected layers of size $(256, 128, 4)$ each with dropout and batch normalization as well. The convolution and linear layers utilize Rectified Linear Units (ReLU) as the primary activation function. The output of the network represents the Q-values of the four possible actions by the agent.

Symmetry To increase generalization and avoid overfitting on a given map, we utilize the symmetrical properties of the Sokoban map and train the network on board symmetries, similar to the methods from



(a) Run time comparison of 60 runs each each across different agents.

(b) Run time comparison of 20 runs each across different files for BoxAgent.

Figure 3: Two different evaluations.

the AlphaGo work[6]. Training on board symmetries ensures that model is generalizing the state-value function for various maps of different orientation rather than overfitting on specific map structures or patterns.

4 Observations and Analysis

In general, we observed the following issues during the design and development process:

i. Deadlock detection is not only non-trivial, but also difficult to exhaustively test.

ii. Determining an effective epsilon greedy rate can be challenging due to long run time and necessary testing.

Moving forward, we refer to the implementation names rather than the algorithm variations; QAgent refers to the traditional Q-learning implementation, BoxAgent refers to the non-traditional Q-learning implementation, and DeepQAgent refers to the deep Q-learning implementation.

4.1 QAgent

QAgent behaved as expected, with slow run times. One particular area that standard Q-learning struggles with is picking sequences of actions from the initial state to partial goal states (i.e., some, not all, boxes on goals). Specifically, the Q-learning algorithm will struggle with long hallways or with fake rooms. Although problematic, this behavior is expected of a random exploration algorithm, as the probability of actions to leave any given room with a single-space exit is $\frac{1}{|S_A|}$, where $|S_A|$ is the length of all combinations of actions with an equivalent sequence length.

4.2 BoxAgent

As seen in Figure 3a, BoxAgent performs faster than QAgent on average, but seems to have a much higher variance. In our personal observation, this run time variances appears sensitive to deadlocks. Figure 3b shows an example of this, where the run time on `sokoban06b.txt` clearly has several large outliers and a much longer inner quartile range. We suspect this variance for this particular file is due to the lack of implementation for corral deadlocks. In general, we suspect the high sensitivity of the performance to deadlocks of each map.

4.3 DeepQAgent

Our DeepQAgent did not converge in training for the general Sokoban solver. We suspect that given the state-action function space required to solve 15×15 boards was too large, resulting in an exorbitant amount of time to train. Moreover, we trained the network on single maps at a time, but for more efficient training, it is critical to train across a variety of different maps at the same time.

In addition to training inputs, exploration strategy might also highly affect the result. We trained the agent with random exploration so each time the agent picks up an action randomly while some actions are obviously invalid. As a result, the agent spent a lot on going back and forth or hitting on the wall. Furthermore, because our agent has higher probability to explore in the early stage, if it does not pick a proper direction early, it might waste tons of time on an unsolvable path or get stuck in the local optimum.

5 Future work

There exists a variety of potential work that can improve our available agents and algorithms. One such example would be an algorithm for identifying corral deadlocks, effectively improving our run time by eliminating further deadlock states.

For the specific agents, BoxAgent can be improved by establishing an ordering on exploration nodes, effectively improving the results of the solution. DeepQAgent in particular holds a lot of interesting possibilities and potential for improvement. Beyond simple convergent training, improving the exploration policy for training and fine tuning the parameter and model size all can be upgraded to reduce training time and increase accuracy.

6 Conclusion

Overall, we implement the traditional Q-Learning algorithm and implement two further improvements utilizing popular techniques in current research efforts. We find that the Q-Learning with a non-traditional state space runs the fastest in comparison standard Q-learning. However, given convergent training we believe that the deep Q-learning model would yield faster performance and a heuristic understanding of the Sokoban game in exchange for lower accuracy, i.e., potentially wrong solutions, and prohibitive training costs. Future work on solving the Sokoban problem would potentially look into lower cost deadlock detection, broader and earlier deadlock detection, parallel Q-learning implementations, and convergent deep Q-learning methods.

References

[1] Alon, N., Avin, C., Koucky, M., Kozma, G., Lotker, Z., and Tuttle, M. R. Many Random Walks Are Faster Than One. 18.

[2] Culberson, J. Sokoban is PSPACE-complete, 1997.

[3] Lin, L.-J. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning 8*, 3-4 (1992), 293–321.

[4] Lovász, L. Random Walks on Graphs: A Survey, Combinatorics, Paul Erdos is Eighty. *Bolyai Soc. Math. Stud. 2* (Jan. 1993).

[5] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. Human-level control through deep reinforcement learning. *Nature 518*, 7540 (Feb. 2015), 529–533.

[6] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. Mastering the game of Go with deep neural networks and tree search. *Nature 529*, 7587 (Jan. 2016), 484–489. Number: 7587 Publisher: Nature Publishing Group.

Appendix

```
jta@Hikari:/mnt/c/users/owner/desktop/home/git/sakoban_ai$ python3 sokoban.py train box benchmarks/sokoban00.txt --time 3600
Create env benchmarks/sokoban00.txt:(3,5) with 1 boxes
1 U
------------------------------
Simulation ended.
time taken       :0.006
solution length  :1
episodes         :11
------------------------------
jta@Hikari:/mnt/c/users/owner/desktop/home/git/sakoban_ai$ python3 sokoban.py train box benchmarks/sokoban01.txt --time 3600
Create env benchmarks/sokoban01.txt:(8,8) with 3 boxes
36 D L L D R D D L L L R R R U U U L L R D R D D L L L U L D U R U U L U R
------------------------------
Simulation ended.
time taken       :9.714
solution length  :36
episodes         :71
------------------------------
jta@Hikari:/mnt/c/users/owner/desktop/home/git/sakoban_ai$ python3 sokoban.py train box benchmarks/sokoban02.txt --time 3600
Create env benchmarks/sokoban02.txt:(8,6) with 3 boxes
41 L L L D L L U R L U U R D D L D R U R R R L L L U U R R R R D D L L L U L U R R R
------------------------------
Simulation ended.
time taken       :69.875
solution length  :41
episodes         :931
------------------------------
jta@Hikari:/mnt/c/users/owner/desktop/home/git/sakoban_ai$ python3 sokoban.py train box benchmarks/sokoban03.txt --time 3600
Create env benchmarks/sokoban03.txt:(8,6) with 4 boxes
52 L L L L L D R U R R R U D L L L L U U R R D U L L D D R U D R U U R R L L D L L U R R R L D D R R U R D
------------------------------
Simulation ended.
time taken       :12.387
solution length  :52
episodes         :131
------------------------------
jta@Hikari:/mnt/c/users/owner/desktop/home/git/sakoban_ai$ python3 sokoban.py train box benchmarks/sokoban04.txt --time 3600
Create env benchmarks/sokoban04.txt:(11,7) with 3 boxes

91 R R R D D L D D L L U R R R R R D R U U U D D L L U L U U L D D U U L L D R U R R D D D D L L U R R R R D R U L L L U U U L L D R U R D R D D R R U D L L L L U R U R D L D R R D R U
------------------------------
Simulation ended.
time taken       :1997.452
solution length  :91
episodes         :3741
------------------------------
```

Figure 4: First 5 benchmarks.

```
jta@Hikari:/mnt/c/users/owner/desktop/home/git/sakoban_ai$ python3 sokoban.py train box benchmarks/sokoban05a.txt --time 3600
Create env benchmarks/sokoban05a.txt:(12,8) with 2 boxes
59 D D D R R R R D D L L U U U R R L L U L U R R L D D R R R L L L U U R R L L D D R R R D R R U L L L L D L U U L U R R
------------------------------
Simulation ended.
time taken       :4.517
solution length  :59
episodes         :21
------------------------------
jta@Hikari:/mnt/c/users/owner/desktop/home/git/sakoban_ai$ python3 sokoban.py train box benchmarks/sokoban05b.txt --time 3600
Create env benchmarks/sokoban05b.txt:(12,8) with 4 boxes
110 R R D D R D R R R U R D D L L L U L L L D D L U R U R R R D D L L U U U L L D D L L U R D R U U U L U R R R D U R R L L D L D D R U U D R R L L U L U R D D R R R L L L U U R L D D R R R D R R U L L L L D
 L U U L U R
------------------------------
Simulation ended.
time taken       :72.552
solution length  :110
episodes         :131
------------------------------
jta@Hikari:/mnt/c/users/owner/desktop/home/git/sakoban_ai$ python3 sokoban.py train box benchmarks/sokoban06a.txt --time 3600
Create env benchmarks/sokoban06a.txt:(15,10) with 1 boxes
40 D L L U U L L L D U R R R D D L L D D D L L U U U R R R R R R R D R U U L U R
------------------------------
Simulation ended.
time taken       :1.382
solution length  :40
episodes         :41
------------------------------
jta@Hikari:/mnt/c/users/owner/desktop/home/git/sakoban_ai$ python3 sokoban.py train box benchmarks/sokoban06b.txt --time 3600
Create env benchmarks/sokoban06b.txt:(15,10) with 3 boxes
108 D L D D D L L L L L U U L U R R R R R R R R R D R U U L U R L D D L L L D D D L L L U U D D L L U U U R R R R R R R R R D R U U L D L L L L U U L L L D U R R R D D L L L D D D L L U U U R R R R R R R R R D R U R U
 L D L U
------------------------------
Simulation ended.
time taken       :6.710
solution length  :108
episodes         :11
------------------------------
jta@Hikari:/mnt/c/users/owner/desktop/home/git/sakoban_ai$ python3 sokoban.py train box benchmarks/sokoban07a.txt --time 3600
Create env benchmarks/sokoban07a.txt:(14,15) with 3 boxes
66 D L L L D D L L D D D R R D R U L U U R U R R D D R R L U U L L L D L D D R D R R R R R U U U D D R U U U D L D L L U U U U R R D R
------------------------------
Simulation ended.
time taken       :977.109
solution length  :66
episodes         :461
------------------------------
```

Figure 5: Benchmarks up to 7a.