# CS271 Project Final Report: Sokoban Solver


**Seokchan Ahn**
15288114
seokchaa@uci.edu

**Youngil Kim**
87561931
youngik2@uci.edu

**Jinhwa Kim**
86727408
jinhwak@uci.edu



## Abstract

Sokoban is a game in which the player pushes boxes around in a warehouse, trying to get them to storage locations. Since the number of possible states can grow exponentially as the board size and number of boxes increase, we suggest model-free reinforcement learning(RL) based solver for this game. Among a variety of RL algorithms, we implement Q-learning to build an efficient solver and suggest the result of our model. By tuning hyperparameters, we show an optimal result for our approach.


## 1 Sokoban

Sokoban is a game that the player pushes the boxes to the proper storage locations with a minimum number of moves. The map is consists of floor, wall, boxes, marked storage locations, and an agent which represents the player. The player can move in 4 directions: Up, Down, Left, Right, and it is not allowed to move through walls or boxes. There are two conditions for terminating the games: 1) All the boxes are on the goals. 2) Deadlocks.

## 2 Q-Learning

Q-learning is a model-free reinforcement learning algorithm to learn the value of an action in a particular state. It is an off-policy learning because the executed actions are different from the target actions that are used for learning. It updates values based on the next state's best action, but choose the action by $\epsilon$-greedy policy, which enables both exploration and exploitation. The convergence of Q-learning is proven (Watkins & Dayan, 1992).

**Algorithm 1:** Q-Learning

**Data:** State $s \in S$ , Action $a \in A(s) = \{U, D, R, L\}$
Initialize $Q(S, A)$ as zero;
Initialize $maxStep, \epsilon, \alpha, \gamma$;
Initialize $s$ ; /* initial state */
$step \leftarrow 1$;
**while** $step \leq maxStep$ **do**
  $a \leftarrow getAction(s, \epsilon)$ ; /* choose action using $\epsilon$-greedy policy */
  $s', r \leftarrow getNextState(action)$ ; /* next state, reward */
  $Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma \max Q(S', a) - Q(s, a))$ ; /* update Q for s, a */
  $s \leftarrow s'$ ; /* update state to new state */
  **if** $isTerminal(s'))$ **then**
    Initialize $s$ ; /* start over from the initial state */
  **end**
  $step \leftarrow step + 1$; /* increase current step by 1 */
**end**

We use $maxStep$ as an upper-bound of the length of each episode. Though Q-learning can learn without limiting the number of steps, having an upper-bound can make the learning process more efficient by preventing exploring the states far from the goal too much. This value must be greater than the length of the episode with the optimal solution since otherwise it will never reach to the goal state. Therefore, we use $maxStep$ of $MNB$ where $M$ and $N$ denotes the board's width and height, and $B$ denotes the number of boxes, since the maximum number of steps to move one box to the storage must be less than $MN$ and there are $B$ such boxes. As a result, the time complexity of this algorithm would be $O(MNB)$ for each episode and total training time would be in proportion to the number of iterations until the Q-values converge. The space complexity would be $O((MN)^{B+1})$ because ignoring the walls, there are $MN$ positions we can place a box or a player, and there are $B$ boxes and 1 player. However, we can save the space a lot in practice by pruning branches that never able to reach a goal.

There are three hyper-parameters in Q-learning. First, $\epsilon$ decides the possibility of exploitation. In general, $\epsilon$ is initialized as a value close to 1.0, which means the agent will choose an action almost randomly, since there is no information about the expected value in the beginning. As the estimated Q-values converge, $\epsilon$ should be decayed to exploit the learned values. $\alpha$ and $\gamma$ are the learning rate and the discount factor by time for each.

**Algorithm 2:** Q-Learning Inference

**Data:** State $s \in S$ , Action $a \in A(s) = \{U, D, R, L\}$
Initialize $s$ ; /* initial state */
$\epsilon \leftarrow 0.0$ ; /* greedy policy */
**while** $True$ **do**
  $a \leftarrow getAction(s, \epsilon)$ ; /* choose action using $\epsilon$-greedy policy */
  $s', r \leftarrow getNextState(action)$ ; /* next state, reward */
  $s \leftarrow s'$ ; /* update state to new state */
  **if** $isGoalState(s')$ **then**
    break ; /* Arrived to one of the goal states */
  **end**
**end**

In the inference time, the agent can greedy choose the best action at the current state every time. It can be done by simply using the $\epsilon$ value of zero. The time complexity would be $O(L)$ where $L$ denotes the length of the episode it generates and $L$ would be less than $M * N * B$.

**Model Selection** In the draft, we propose three different algorithms: Monte-Carlo Learning, Temporal Difference Learning, and Q-learning. Each algorithm has different advantages and disadvantages.

*Monte Carlo Learning* is one of the simple learning methods. It is simple and easy to implement. However, it only uses terminated episodes for updating the value function, so it could be slower than other algorithms. The advantage is it does not suffer from bias.

*Temporal Difference Learning* updates the value function after every step even it is not terminated. It could be faster than the Monte Carlo method, but it causes high bias.

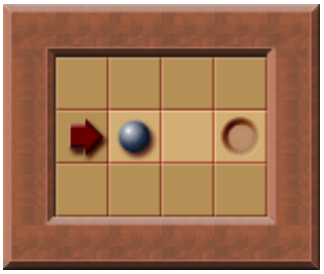
*Q-learning* is an off-policy method. It gathers information from partially random moves and slowly reduces the randomness. Thus, it could be slower than the on-policy method, but it would be more flexible if alternative routes appear.

We want our model to be flexible in learning not only in simple cases, but also in a diverse and complicated cases. Thus, we select Q-learning as the model of our design.

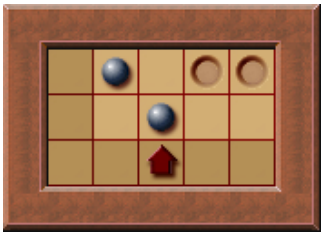
## 3 Implementation Details

### 3.1 States and Actions

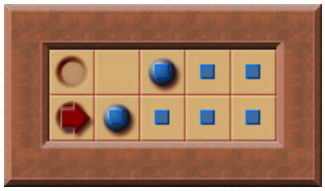
In the Sokoban game, states can be defined as the setting of the board. Since the positions of walls and the positions of storage are fixed, we can use the coordinations of the boxes and the player as a state. Since the order of boxes doesn't matter, we use a hash map for Q-table using $(set(boxes), player)$



(a) Simple Deadlock



(b) Freeze Deadlock



(c) Corral Deadlock

Figure 1: Three simple graphs

as keys and Q-values as values. For actions, we use $[L, U, R, D]$ since they are all the available movements.

A Sokoban game's terminal states are 1) when all the boxes are on the storage or 2) when on a Deadlock state. In training time, we stop and re-initialize the state once the player reaches to a terminal state since we do not need to explore further. In inference time, we just stop and return the generated episode and its length.

#### 3.1.1 Deadlocks

In Sokoban games, there is a state, which is called a deadlock, when an agent cannot reach any goal state, no matter what the agent moves. The only way to solve this problem is backtracking to the previous state or restarting the game. In order to solve the Sokoban game efficiently, it is inevitable to handle deadlocks properly. In our project, we handle only simple deadlocks to prune unnecessary branches in advance, but we decide not to take further actions because it is sufficient to solve Sokoban game with solely simple deadlocks detection as well as there is an overhead to detect other complicated types of deadlocks. [1]

**Simple deadlocks** A simple deadlock is created by moving just one box to border lines, which makes it unable to reach any goal state, no matter what the agent moves. In the figure 1a, pushing the box to a darker shaded space results in a simple deadlock.

**Freeze deadlocks** Freeze deadlocks refer to the situation that one of boxes becomes immovable. If a box becomes immovable while not being located on a goal, the game is deadlocked. In the figure 1b, pushing the box one square up results in a freeze deadlock.

**Corral deadlocks** Corral indicates a zone that the agent cannot access. Corral deadlocks can be created when deadlock happens if the agent is not able to move the box because of presence of corral. In the figure 1c, marked area with little blue quadrats is a corral, and it is not reachable for the agent. Pushing the lower box to the right square results in a corral deadlock.

### 3.2 Rewards

Rewards are the most important part in reinforcement learning since it decides which actions are good or bad and how good or bad they are. We defined the categories of rewards as below.

[1] http://sokobano.de/wiki/index.php?title=Deadlocks

| Category | Reward | Description |
|---|---|---|
| MOVE | -0.1 | To prevent looping infinitely, give small penalty for each move |
| BOX_ON_STORAGE | 1.0 | Since it may takes too many steps to the final goal, updating Q-values from the final reward is difficult. Therefore, giving rewards for achieving intermediate goals can make the Q converge faster. |
| BOX_OFF_STORAGE | -1.0 | Give penalty for pushing a box away from a storage. |
| ALL_ON_STORAGE | 10.0 | Give reward when the player successfully finishes the game. |
| DEADLOCK | -10.0 | Once it gets to a deadlock state, it can never be solved afterwards. Therefore, give a great penalty and initialize the state again. |
| MOVE_TO_WALL | -1.0 | Trying to go toward a wall or a box in front of another box or a wall is not worth. Therefore, give a small penalty. |

Table 1: Categories of rewards and their values

# 4 Experiments

## 4.1 Hyperparameter Optimization

A hyperparameter is a parameter whose value is used to control the learning process. These parameters are tunable and they could directly affect to the model performance. Thus, optimizing those parameters is inevitable to maximize the performance. We explore the hyperparmeter space by grid search to figure out the optimal hyperparameters.

**Learning rate** $\alpha$ The learning rate is used to determine to what extent newly acquired information overrides old information. When the learning rate is set to 0, the agent learn nothing. In case of 1, update current Q-value by the amount of TD-error. We experiment tuning the learning rate from 0 to 1 in increments of 0.2 for each step.

**Discount factor** $\gamma$ The discount factor is used to determine the importance of future rewards. If the discount factor is set to 0, do not consider the next Q-value at all and with the discount factor of 1, fully attend the next step's Q-value. We experiment tuning the discount factor from 0 to 1 in increments of 0.2 for each step.

**$\epsilon$-decay rate** The $\epsilon$-decay rate decides how much to reduce the $\epsilon$ for every successful episode. We need to decide $\epsilon$-decay rate between 0 and 1 because we need to increase the possibility of exploitation over exploration as learn more by decreasing $\epsilon$. We experiment varying $\epsilon$-decay rate from 0.5 to 0.9 in increments of 0.1 for each step. Initially, we set $\epsilon$ as 0.9, and it is adjusted with the $\epsilon$-decay rate during the iteration.

Since we use $\epsilon$-greedy method, which is a stochastic component, the policy can be varied for each training even with the same hyperparameters. Thus, We iterate training 5 times and choose the best hyperparameters based on the mean of the results.

To discover the optimal hyperparameters, we evaluate the learned Q-values by three criteria. First, we check whether the Q-learning is converged. We define convergence as the greedy policy stalls for 5 consecutive evaluations during the training process. To expedite experiments, when the number of iteration exceeds a specific number, 100,000, we consider it as diverging and stop learning. Second, we compare the length of the episode when following the greedy policy. We prioritize the Q that generates shorter episode. Finally, we prefer the parameter that converges in shorter number of updates.

In the experiment, we use the input data provided by the class, $sokoban01.txt$, because $sokoban00.txt$ is too trivial to measure performance of the algorithm.

## 5 Results

| Learning rate $\alpha$ | Discount factor $\gamma$ | $\epsilon$-decay rate | Converged? | length of episode | # of steps |
|---|---|---|---|---|---|
| 0.2 | 0.2 | 0.5 | No | - | - |
| 0.2 | 0.2 | 0.7 | No | - | - |
| 0.2 | 0.2 | 0.9 | No | - | - |
| 0.2 | 1.0 | 0.5 | Yes | 22 | 74,400 |
| 0.2 | 1.0 | 0.7 | Yes | 22 | 51,620 |
| 0.2 | 1.0 | 0.9 | Yes | 22 | 62,100 |
| 0.6 | 0.6 | 0.5 | Yes | 36 | 39,440 |
| 0.6 | 0.6 | 0.7 | Yes | 36 | 42,480 |
| 0.6 | 0.6 | 0.9 | Yes | 36 | 62,450 |
| 1.0 | 0.2 | 0.5 | Yes | 36 | 36,020 |
| 1.0 | 0.2 | 0.7 | Yes | 36 | 49,820 |
| 1.0 | 0.2 | 0.9 | Yes | 36 | 42,760 |
| 1.0 | 1.0 | 0.5 | Yes | 22 | 25,960 |
| 1.0 | 1.0 | 0.7 | Yes | 22 | 23,740 |
| 1.0 | 1.0 | 0.9 | Yes | 22 | 29,800 |

Table 2: Parameter Tuning Results

Table 2, shows a partial results of hyperparameter exploration. We have found that the algorithm does not work when both the learning rate $\alpha$ and the discount factor $\gamma$ are low, regardless of the $\epsilon$-decay rate. Also, surprisingly, the algorithm tends to find a non-optimal episode when either the learning rate $\alpha$ or the discount factor $\gamma$ is high.

The Q-learning converged to the optimal policy the fastest after 23,740 iterations, with the learning rate $\alpha$ of **1.0**, the discount factor $\gamma$ of **1.0**, and the $\epsilon$-decay rate of **0.7**.

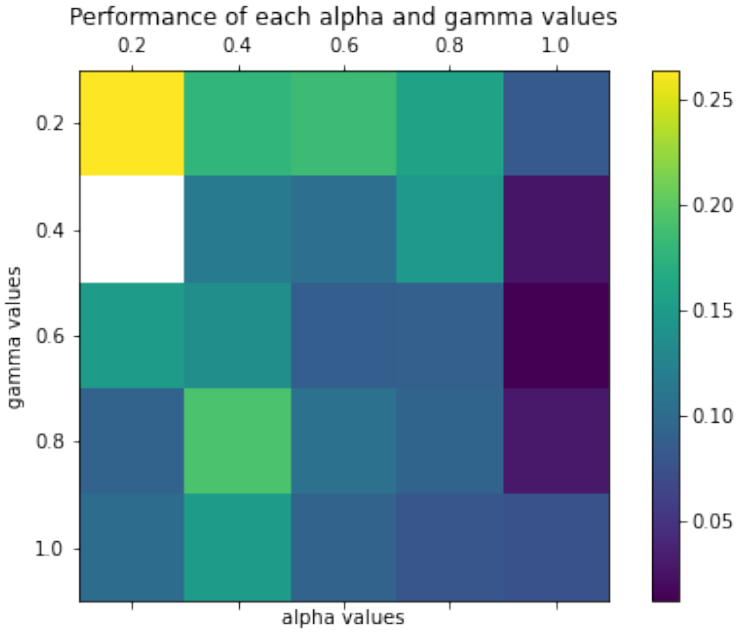



Figure 2: Performance on each learning rate $\alpha$ and discount factor $\gamma$

**Comparison of performance for two hyperparameters** To find the relationship between hyperparameter setting and performance, we plot each result in the respect of two hyper parameter values: $\alpha$ and $\gamma$. We consider two factors, *length of episode* and *number of steps*, as the performance of the model and normalize each of the values and calculate the weighted sum of the two values. Since the

length of episode is a more important factor, we give 0.8 as a weight for the length of the episode and 0.2 for the number of steps. Figure 2 shows the result of performance and the darker square means more optimal result and white square means not converged case. The result shows that higher $\alpha$ tends to make a better result regardless of the value of $\gamma$.

# 6 Analysis

## 6.1 Consideration for rewards

In this section, we evaluate various rewards we set in Chapter 3, to figure out the impact of individual rewards by getting rid of one of them respectively. To get a stable result, we iterate the algorithm 5 times for each time due to the stochastic characteristic of the algorithm.

**MOVE** We expect that the algorithm will be stuck on infinite loops if we remove the reward for $MOVE$, not being able to find a solution of the game. As we have expected, the algorithm cannot reach a goal state after 100,000 steps we set as a maximum because it may repeat redundant steps. Thus, it is necessary to assign a negative reward for moving each step.

**BOX_ON_STORAGE** For the input $sokoban01.txt$, the algorithm never arrive to the goal state without the reward $BOX\_ON\_STORAGE$. The algorithm works without an intermediate reward in a trivial case like $sokoban00.txt$, but in the larger problems, a guidance reward is necessary because as the minimum steps to the goal grows, the possibility of arrival decreases exponentially with the random moves and it will never get a chance to increase state-action values.

**BOX_OFF_STORAGE** We added the $BOX\_OFF\_STORAGE$ reward because we expected moving a box away from a storage will make the state away from the goal state. But, in the $sokoban00.txt$ case, there was no significant difference in the results after removing the $BOX\_OFF\_STORAGE$ reword. This might be because in the given example, the agent do not push away the box once the box is moved to the storage when following the greedy policy.

**ALL_ON_STORAGE** We give a reward $ALL\_ON\_STORAGE$ when the player succeeds to locate all boxes to storage to encourage the agent move to this final goal state. However, since the agent can still get rewarded from $ALL\_ON\_STORAGE$ but from $BOX\_ON\_STORAGE$, the algorithm can still find a solution successfully without $ALL\_ON\_STORAGE$ reward. We expect this reward would help finding the goal state a little faster.

**DEADLOCK** We introduced $DEADLOCK$ reward because the algorithm sometimes repeated moving in a small area infinitely without this. By penalizing the deadlock state, the algorithm converged faster and found the solution by avoiding the states that have no chance to reach the goal.

**MOVE_TO_WALL** We add $MOVE\_TO\_WALL$ reward to eliminate unnecessary attempts to move to walls. Our experiment shows that the presence of the reward reduce the total number of steps to the goal by 15%, compared to the case without this reward.

## 6.2 Limitation of our approach

We make use of Q-learning algorithm to find an optimal solution to the given Sokoban game, but the algorithm's scalability is limited since the number of states grow exponentially with the size of the board and it will consume exponential time and space. As an alternative, we may use another approach such as Deep Q Network (DQN) learning, which approximates the Q function using the fixed number of parameters.

# 7 Conclusion

In this project, we defined the states, actions, and rewards of the Sokoban game to solve the problem as a reinforcement learning problem, and implemented a Sokoban solver using Q-learning algorithm. For hyperparameter search, we use grid search for three hyperparameters, $\alpha$, $\gamma$, and $\epsilon$. And we

averaged the best parameters after running 5 times, since the Q-learning contains $\epsilon$-greedy method, a stochastic component. The best hyperparmeters found for the $sokoban01.txt$ problem were the learning rate $\alpha$ of **1.0**, the discount factor $\gamma$ of **1.0**, and the $\epsilon$-decay rate of **0.7**. With that settings, the length of the episode to the goal state was 22 and it took 23,740 iterations to be converged in average.

## 8 References

[1] Watkins & Dayan (1992). Q-Learning, Machine Learning, 8, 279-292.

[2] Russell, S. J., Norvig, P., & Davis, E. (2010). Artificial intelligence: a modern approach. 4th ed. Upper Saddle River, NJ: Prentice Hall.