# Solving Sokoban Using Q-Learning

Joseph Herkness
Oakland University
jpherkness@oakland.edu

Joshua Herkness
Oakland University
jrherkness@oakland.edu

## Abstract

*This paper describes a Q-Learning based algorithm used to determine the solution to a sokoban puzzle. The nature of Sokoban, as a finite decision problem, means that a Reinforcement Learning approach can be used to identify the problems optimal policy. Q-Learning is a Reinforcement Learning technique that constructs the optimal policy through recording the utility of executing an action on a given state.*

## 1. Introduction

Q-Learning is a form of model-free reinforcement learning that provides agents with the capability to develop a strategy for solving an instance of a problem. Model-free learning differs from model-based learning in that it does not require a pre constructed model of the problem in order to derive a solution. Instead, Q-Learning derives the solution through exploring the problem space, much like many dynamic programming concepts. In this paper, a Q-Learning algorithm, that identifies the solution to a Sokoban puzzle, will be explained. A Sokoban puzzle is a grid based logic puzzle where a player pushes boxes around a warehouse, the ultimate goal being to position these boxes on top of every goal. The typical Q-Learning approach will be improved upon through an analysis of potential traps in each state, as well as adjustments to the learning rate and discount factor through dynamic modification as episodes progress.

### 1.1. Motivation

Sokoban has been proven to be a PSPACE-Complete problem [1]. Other approaches to this problem include BFS, DFS, and solvers employing heuristic functions such as A* search.

The problem of solving a sokoban puzzle is often compared to the real world problem of programming an autonomous robot to work in a warehouse. Such a robot would be required to navigate the warehouse, as well as perform its designated tasks, using only the information in its immediate vicinity. If such a robots task were to move crates into a storage location, then the two problems would be synonymous. In this case, a similar solution could be used to solve both problems. If Q-Learning proves to be a valid solution to the sokoban problem, its application to real world problems could prove to be limitless.

If, however, there was more than a single agent acting within the environment, the basic Q-Learning approach will fail. For example, imagine a situation where multiple autonomous robots are each moving boxes within a warehouse from location to location. If a basic Q-Learning approach was implemented, these robots would begin to interfere with one another. To fix this, the robots would need some way to communicate learned information to and from one another [7].

## 2. Sokoban

Sokoban is two dimensional puzzle game. Each Sokoban level consists of a rectangular grid representing a warehouse. The warehouse contains many different entities, such as boxes, goals, walls, and the player themselves. Each entity is restricted to the two dimensional grid of the warehouse. In order to solve the puzzle, the player must push each box such that it covers one of the goals. When every goal has been covered by a box, the puzzle is considered solved. A valid Sokoban puzzle must contain a number of boxes equal to the number of goals. Each action that the player takes has the potential of placing the level in a deadlock. If a deadlock is created, the level is no longer solvable, and the player has lost.

The level shown in figure 1 is the simplest solvable sokoban level that can be made. The state on the right is the initial state, and the state on the left is the final state. The solution to the puzzle is to simply move to the right one space. This action pushes the box onto the goal, solving the level.

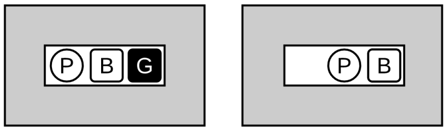

**Figure 1. The simplest solvable sokoban level (left) and the solved state (right).**

### 2.1. Rules

During each turn, the player has the ability to move in one of four directions - up, down, left, or right. A square is considered empty if it does not contain a wall or box. If the square corresponding to the direction moved contains a box, and the next square is considered empty, the box is pushed into that square. The player is allowed to move into any empty square. When every goal has been covered by a box, the puzzle is considered solved, and the game ends.

### 2.2. Deadlocks

A deadlock is a configuration of boxes that results in an unsolvable level [1]. A state is considered deadlocked if it contains at least one deadlock. Any action that the player takes has the potential of placing the level in a deadlocked state. If such a state is encountered, any attempt to solve the level should no longer be pursued, even if there are still valid actions that can be applied to the deadlocked state. While there are numerous types of deadlocks, two that are necessary for complete deadlock detection are simple deadlocks and freeze deadlocks. Implementation of the detection of these two deadlock types results in the ability to detect whether or not a action creates a deadlocked state.

### 2.3. Simple Deadlocks

Simple deadlocks are squares in the level that create a deadlock whenever a box is pushed into them. Pushing a block into a simple deadlock square creates a deadlock because the box in that square can no longer be pushed to a goal. These simple deadlock squares are independent of the positions of the boxes in the level, and do not change. This means that simple deadlock squares can be identified when the initial level is loaded.

The algorithm for identifying simple deadlock squares is rather intuitive. By definition, if a box cannot be pushed from a square to one of the goals, the square is considered to be a simple deadlock square. Likewise, if a box cannot be pulled from one of the goals to that square, it means the same thing. This mean that all valid squares can be identified by removing all boxes from the level, placing a box on each goal square, pulling the box away in all directions, and marking all reachable squares as visited. Every square that is not marked as visited is a simple deadlock square.

If we consider the level in figure 2, where the simple deadlock squares are crossed off, the algorithm becomes a little more clear. First we remove all boxes from the level. We then place an imaginary box on the goal and pull it in each direction. Since we cannot pull the box up, the space directly above the goal is a simple deadlock space. Since we cannot pull the box down, the square directly below the goal is a simple deadlock square. However, since we can pull the box to the left, the square directly to the left of the goal is considered a valid square. This square is then marked as visited, and we repeat the algorithm on this new square.

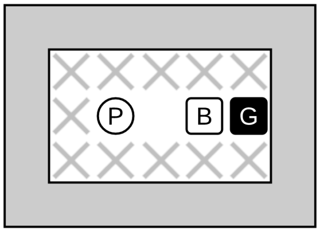

**Figure 2. The simple deadlock squares are identified in this level and marked.**

The result of performing such an algorithm is the identification of all simple deadlock squares. When a box is moved, we can check the see if the square it is moved into is a simple deadlock square. If it is a simple deadlock square, we know there is a deadlock and the level can no longer be solved.

**Algorithm 1** Identifying simple deadlocks

```
function IDENTIFYSIMPLEDEADLOCKS
    $stack \leftarrow$ goals
    $visited \leftarrow$ empty set
    while stack not empty do
        $position \leftarrow stack.pop()$
        visited.add(position)
        for direction = up, down, left, right do
            if can pull position in direction then
                $valid \leftarrow$ move position in direction
                stack.add(position)
            end if
        end for
    end while
    return all squares - visited
end function
```

### 2.4. Freeze Deadlocks

Freeze deadlocks are configurations of boxes and walls that result in a deadlocked state. Unlike the simple deadlock squares, freeze deadlocks depend on the position of boxes in the level. This means that we must check for a freeze deadlock anytime a box is moved in the level.

The level in figure 3 demonstrates an example of a freeze state. Notice how if either box is pushed up, it will be moved into a simple deadlock square. Since we can not push either box up, they are both blocked along the vertical axis. Since the blocks are side by side, any attempt to push the boxes left or right will fail. This mean each box is blocked along the vertical axis. Since both boxes are blocked along the vertical and horizontal axis, the entire configuration is considered a freeze deadlock, meaning no box in the configuration can be moved. However, if all boxes in the configuration are on goals, the level is considered to be in a semi-solved state, and there is no freeze deadlock. An example of such a configuration can be seen in figure 4.

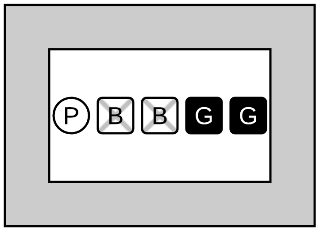

**Figure 3. The configuration of boxes in this level creates a freeze deadlock.**

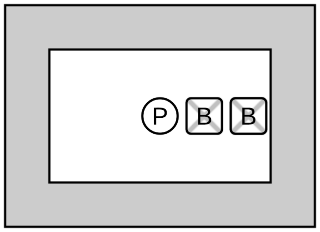

**Figure 4. Since the boxes in this level are frozen on top of the goals, no freeze deadlock is created.**

The algorithm for detecting freeze deadlocks is rather intuitive, since we only need to check whether a box can be pushed. Since a freeze deadlock is created after pushing a block, it is only neccessary to check if the pushed box is frozen, and possibly the boxes around the pushed box. The following algorithm will identify whether or not pushing a box into a position will create a freeze deadlock.

**Algorithm 2** Identifying freeze deadlocks

```
$frozen$ ← empty set
$visited$ ← empty set
function FROZEN(position)
    add position to visited
    if wall on left or right then
        $h \leftarrow true$
    else if simple deadlock on on left and right then
        $h \leftarrow true$
    else if box on left or right then
        $h$ ← Frozen(left) or Frozen(right)
    end if
    if wall on up or down then
        $v \leftarrow true$
    else if simple deadlock on on up and down then
        $v \leftarrow true$
    else if box on up or down then
        $v$ ← Frozen(up) or Frozen(down)
    end if
    return h and v and all frozen on goals
end function
```

### 2.5. Notation

## 3. Related Work

## 4. Q-Learning

Q-Learning is one of the more prominent reinforcement learning techniques. This techniques can be used to find the optimal actions that should be taken for any given markov decision process. A markov dicision process is a mathematical framework used to model decision making problems where the decision are under the agent's control. At any given state $s$ the agent can perform an action $a$. The result of taking action $a$ is the new state $s\prime$. The agent is then awarded a reward of $R_a(s, s\prime)$ for taking action $a$. The agents goal is to develop a strategy for solving the problem that results in the highest reward.

Q-Learning allows a single agent, aware of its current state, to develop a strategy through exploration. This is accomplished through consideration of past states and future reward for performing all possible actions on the current state. An agent is placed into its environment at an initial state $s$ and begins by taking some arbitrary action $a$. Once this action is taken, the Q-value for the given state (s), after taking that action (a), is updated. The agent is said to have learned from this action (a), and will use this learned reward in the future if it encounters this state (s) again. The Q-value for taking an action (a) at a state (s) is defined as [**?**]:

$$\begin{aligned} Q(s_t, a_t) \leftarrow & Q(s_t, a_t) \\ & + \alpha \cdot [R_a(s_t, s_{t+1}) \\ & + \gamma \cdot max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \end{aligned} \tag{1}$$

Where  is the Learning Rate of the agent, and  is the Discount Factor of the agent.

## 4.1. Rewards

Rewards are one of the major building blocks of Q-Learning. If you recall, a reward is given to an agent once an action $a$ is perfomed on a state $s_t$. Many Q-Learning algorithms implement a reward function which, depending on the outcome state $s_{t+1}$, will result in a numberical reward. These rewards will fall into either one of two categories, negative or positive. A positive reward should be given when a desirable outcome is acheived after action $a_t$ is performed at state $s_t$, resulting in an outcome state $s_{t+1}$. The nature of the rewards implementation in the Q-Learning algorithm will cause the agent to perfer taking action $a$ at state $s_t$ again. A negative reward should be given when a undesirable outcome is acheived after action $a$ is performed at state $s_t$, resulting in an outcome state $s_{t+1}$. Converse to the effect that the positive reward will have, the negative reward will cause the agent to avoid taking action $a$ at state $s_t$ again. The formula for this reward is generally written as a difference function between the intial state $s_t$ and outcome state $s_{t+1}$ as seen in equation 2.

$$reward \leftarrow R_a(s_t, s_{t+1}) \tag{2}$$

A full list of the finalized rewards in our soluction can be found in table 1. We found that positive rewards should be given very rarely, and only in extremely desirable circumstances. For instance, we gave the highest (positive) reward when a box was pushed onto a goal. In the inital design of our solution, we gave the agent a reward of 0 for moving, because we saw no immediate negative or positive effects from this. However, in our final implementation, we gave a negative reward to an agent for moving because this encouraged exploration and discouraged 'wondering'. In the development of a successful Q-Learning algorithm for the Sokoban puzzle, we found that the adjustment of our rewards had a significant impact on the success rate of the solver.

## 4.2. Algorithm

| Reward Value | Reward Type |
|---|---|
| +1.00 | Box pushed onto goal |
| -0.01 | Box pushed |
| -0.15 | Agent moved |
| -0.15 | Box pushed against wall |
| -0.15 | Box pushed against box |
| -0.15 | Box pushed from one goal to another |
| -0.20 | Box pushed off goal |
| -1.00 | Agent moved into wall |
| -1.00 | Agent moved into immovable box |
| -1.00 | Agent created simple deadlock |
| -1.00 | Agent created freeze deadlock |

**Table 1. Rewards awarded to the Sokoban agent during Q-Learning**

The following Q-Learning algorithm [3][6] will be used to determine the optimal strategy for solving a given Sokoban puzzle. In the algorithm, an episode represents one iteration from initial state to terminal state. Numerous episodes will be executed in sequence, allowing the agent to learn which actions produce the greatest reward. The greater the number of episodes, the more optimal the solution will become. When the q-values converge, meaning they stop changing between episodes, the optimal solution will be found.

One of the most critical aspects of the algorithm is identifying if the current state is a terminal state. In the case of Sokoban, a terminal state is defined as a state where the puzzle is solved, or a state that is no longer solvable. The following predicates will be used to [1] analyze the state and determine if it has reached a terminal state.

**Algorithm 3** Solver for Sokoban using q learning

```
function QLEARNING(state, episodes)
    $Q \leftarrow$ empty set
    for episode = episodes do
        $Q \leftarrow$ RunEpisode(Q)
    end forreturn Q
end function
```

**Algorithm 4** Runs a single episode of q learning

```
function RUNEPISODE(state, Q)
    $state \leftarrow$ initial state
    while not Terminal?(state) do
        $action \leftarrow$ MaximizeAction(state, Q)
        $resultState \leftarrow$ TakeAction(state, action)
        update Q value
    end while
    return Q
end function
```

**Algorithm 5** Returns the maximum q value reachable from a state in a single action

```
function MAXIMIZEQ(state, Q)
    $max \leftarrow -\infty$
    for action = up, right, down, left do
        if $Q(state, action) > max$ then
            $max \leftarrow Q(state, action)$
        end if
    end for
    return $max$
end function
```

**Algorithm 6** Returns the action which will acheive the maximum q value reachable from a state

```
function MAXIMIZEACTION(state, Q)
    $max \leftarrow -\infty$
    $actions \leftarrow$ empty set
    for action = up, right, down, left do
        if $Q(state, action) > max$ then
            $max \leftarrow$ Q(state, action)
            $actions \leftarrow$ set with action
        else
            $actions \leftarrow actions + action$
        end if
    end for
    return random action from $actions$
end function
```

**Algorithm 7** Returns the resulting state after an action is taken on an intial state

```
function MAXIMIZEQ(state, action)
    return state after action is taken
end function
```

## 5. Experimental Analysis

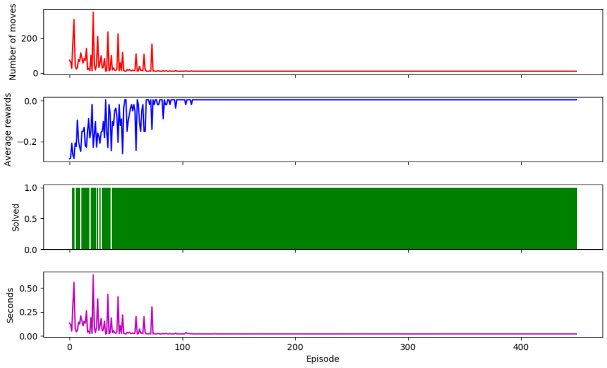

**Figure 5. Results for solving an easy level.**

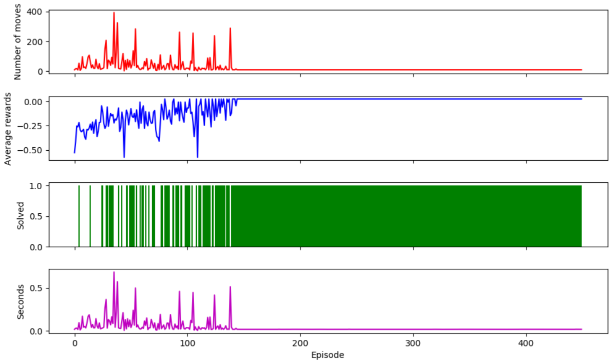

**Figure 6. Results for solving an medium level.**

### 5.1. Rewards

### 5.2. Learning Rate and Discount Factor

## 6. Conclusions

## References

[1] J. Culberson. *Sokoban is PSPACE-complete*. 1997.

[2] P. Jaruek and R. Pelnek. Difficulty rating of sokoban puzzle in Proc. of the Fifth Starting AI Researchers' Symposium (STAIRS 2010). Dec, 2010. pp. 140-150

[3] S. J. Russell and P. Norvig. *Artificial intelligence: a modern approach*. Upper Saddle River: Prentice-Hall, 2010.

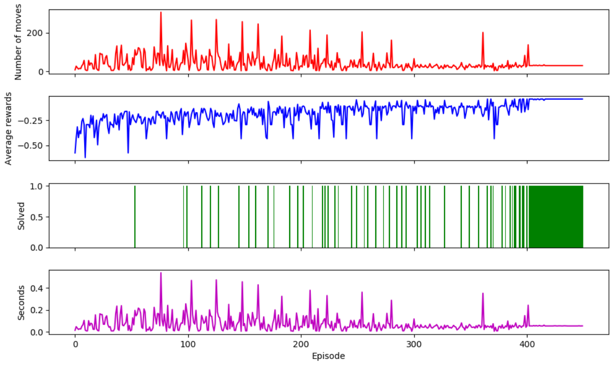

**Figure 7. Results for solving an hard level.**

[4] M. L. Littman. Markov games as a framework for multi-agent reinforcement learning in Proceedings of the eleventh international conference on machine learning. Vol 157. pp 157-163. 1994

[5] M. Guo, Y. Liu and J. Malec *A new Q-learning algorithm based on the metropolis criterion*. IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics), vol. 34, no. 5, pp. 2140-2143, Oct. 2004.

[6] T. Ming. *Multi-agent reinforcement learning: Independent vs. cooperative agents.*. Proceedings of the tenth international conference on machine learning. 1993.