Reinforcement Learning and Sokoban

Ethan Heavey
St. Francis Xavier University

1 Introduction

Markov Decision Processes (MDP's) are a class of mathematical models that formalize sequential decision-making [4], observing the current state of the system and the current agent, and how those current states affect the cost and transition functions [3]. Reinforcement Learning (RL) can be applied to many different fields, economics, medicine, and the automotive industry to name a few. This field of computer science, due to the sequential nature of how RL models can solve problems, is idealized by structuring problems through the lens of the Markov Decision Process (MDP). A MDP is a mathematical model that formalizes sequential decision making, allowing it to work hand in hand with RL models.

This report seeks to employ RL and MDP on the Sokoban game, in an attempt to have an agent win the game. By framing the game as a MDP problem, we can develop a RL algorithm that will attempt to "win" Sokoban.

2 Markov Decision Process

2.1 Definition

As a mathematical model that formalizes sequential decision making using last results to predict future ones, a MDP is a powerful tool to make RL work in our favour. Due to the nature of MDP, we know the problem perfectly and try to find a solution, which, through feedback from the environment, is what the agent works towards.

A MDP consists of an agent and the environment surrounding the agent, and is formalized in a four-item tuple; MDP = (S, A, T, R), where:

- S is the finite set of states, where $s \in S$
- A is the finite set of actions, where $a \in A$
- T: SxAx$S \to [0,1]$ is the transition function

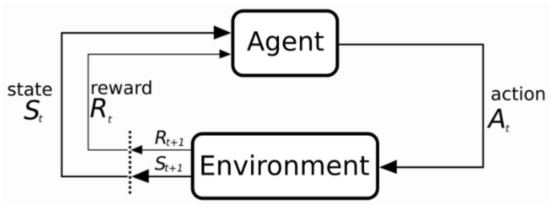


Fig. 1: A flowchart of a MDP cycle

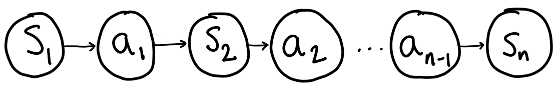


Fig. 2: An example of a trajectory in an MDP environment

- R: Sx$A \to \mathbf{R}$ is the reward function

Figure 1[1] represents the interaction between the agent and environment. At any given time step t, the agent must base their action, a_t, on the current state of the environment, s_t. After the agent executes its chosen action, the environment gets updated to state s_{t+1}, and the agent receives a reward, R_t, from the environment. This cycle of events yields a trajectory (a sequence of states, actions, and rewards) akin to the one depicted in Figure 2.

2.2 Sokoban MDP

To frame Sokoban as a MDP problem, we need to identify all possible states (S), all possible actions (A), the transition function (T), and the reward function (R). This implementation of a Sokoban gym[1] environment also relies on the maximum number of steps the agent is allowed to take, with this parameter being an alternate ending condition of the simulation.

Each state in the game must contain the following; the location of the agent and the location of

[1]https://people.stfx.ca/jdelamer/courses/csci-531/topics/markov-decison-process/markov_decision_process.html

Attribute	Example
agent_location	$[2, 2]$
box_location	$[[0, 2], [3, 5], [1, 3]]$

Table 1: A Table showing examples of the elements comprising each state.

each box, and is structured as follows:

$$(agent\_location, box\_locations)$$

The agent location will be a tuple containing two values, the x and y position of the agent in the space. Box locations will be a list of two-element tuples detailing the locations of all the boxes in the level. Similar to box locations, goal locations will be a list of two-element tuples. Box locations and goal locations are lists of the same length. Examples of each attribute are depicted in Table 1.

An important observation regarding the state and the agent's perception of said state is that that if two boxes are in adjacent tiles, they cannot both be pushed together. A box can only be moved if there is an empty space in the direction the agent is attempting to push it.

There are four possible actions the agent can take, move up, move down, move left, and move right. The current state will dictate what actions the agent can make, For example, if there is a wall to the agent's right, and two boxes in both tiles directly below the agent, the agent can only move left or up.

The transition function comes into play once the agent executes an action. Upon execution, the agent's chosen action will have an impact on the current state s_t and bring us to s_{t+1}. When the player moves, the next state will contain the player's updated position, and the updated position of the box the player pushed, if it pushed any. The probability of s_{t+1} occurring is 1 if the action given to the state is legal and results in s going to s_{t+1}, and 0 otherwise. The walls, and the final locations of boxes, will never be moved and have a permanent position in the environment.

When an agent executes an action, it receives a reward. In of the Sokoban game, the goal is, after the agent's final action, to have all the boxes be in an "end tile" - or the tiles that all must be have a box in them for the game to be considered "won." The reward function that can be used for this game is simple, give the agent a score of -0.001 for every action it makes, as if the priority of the agent is to get all boxes onto an end tile as fast as possible, the aforementioned negative reward (or penalty) to each move will accomplish this. While the objective is to get all the boxes to an end tile, if the agent places a box on a tile and receives a reward greater than what it receives for moving a box off of an end tile (for example, $R_t = 10$ if moved onto a tile, -5 if moved off), the agent may choose never to remove a box from a tile. To avoid this trap, the absolute value of the penalty for moving boxes off of tiles will be less than the absolute value of the reward for moving a box onto a tile. The Agent will receive a reward valued at 0 if all the boxes are on goal tiles before they run out of steps, capped at 100.

Character	Meaning
#	Wall
.	Empty Space
A	Agent Start Location
B	Box Start Location
G	Goal Location
$	Box-on-Goal

Table 2: A Table depicting the symbols to be used in a Sokoban level that will be loaded into the environment.

2.2.1 Loading the Level

The level can be loaded from any `.txt` file with the symbols outlined in Table 2. An important not when creating a custom level, if a level is of length l and width w, the `.txt` file must fill all $l * w$ spaces with one of the symbols. The outer edges should all be composed of # characters, representing the outer boarder of the level. The default level is shown in Figure 3.

When loading the project, ensure the directory structure is as follows prior to execution:

```
/
├── sokoban.py
└── Levels
    ├── level_0.txt
    ├── level_1.txt
    └── All other custom levels
```

Feel free to create and add custom levels! When adding and executing the script, it will print the starting state, and then the states after executing a sequence of actions.

3 Reinforcement Learning Algorithm

To aid the agent in it's attempt to solve Sokoban, we will be implementing the on-policy semi-gradient SARSA. This algorithm is an extension of TD(0) and instead of updating a policy directly, SARSA updates a weight vector using the following formula:

$$w_{t+1} = w_t + \alpha[U_t - \hat{\boldsymbol{q}}(s_t, a_t, w_t)]\nabla\hat{\boldsymbol{q}}(s_t, a_t, w_t)$$

where U_t can be any approximation of $q_\pi(s_t, a_t)$. The episodic semi-gradient one-step SARSA is the variant of the algorithm being implemented, which changes U_t in the above equation to

$$U_t = r_t + \gamma\hat{\boldsymbol{q}}(s_{t+1}, a_{t+1}, w_t)$$

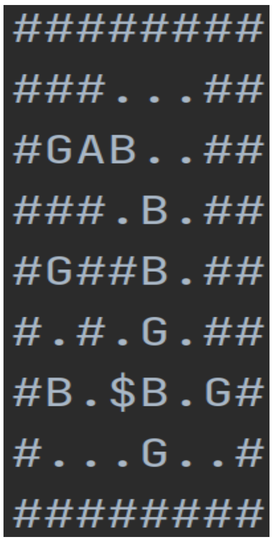
The formula now computes $\hat{\boldsymbol{q}}(s_{t+1}, a_t, w_t)$ for every action a, finds the greedy action, and changes the estimation policy to a soft approximation of the greedy policy. It is important to note that even though the action is passed to the formula, it is not considered when calculating the dot product.

To incorporate all elements of the state into the weights, we must create the $X(s)$ vector where it is the same length as the weight vector w. For this implementation of semi-gradient SARSA, the length of both vectors will be 4. The $X(s)$ will be defined as follows:

$$X(s) = [1, x_{agent}, y_{agent}, x_{box^i}, y_{box^i}]$$

where x and y are the coordinates of the agent and all boxes. Note that prior to calculating the $X(s)$ vector for a given state s, we will multiply the coordinates element-wise by the location of the agent, thus incorporating all aspects of the state when calculating the weights. With this problem we will be assuming there is a linear approximation function, thus the derivative of $\hat{\boldsymbol{q}}(s_{t+1}, a_t, w_t)$ will be the dot product of the weight vector and the $X(s)$ vector.

Initially, the state was represented as an integer, with each pair of digits within the number representing the location of the agent and each box. This proved to be a major issue for training as the weight vector would diverge to positive or negative infinity with almost every execution. After observing these trends, the state and state vector were altered to their current state outlined above. The old state vector contained a measly three features, which is insufficient when attempting to investigate the many, many states Sokoban has to offer.



```
########
###...##
#GAB..##
###.B.##
#G##B.##
#.#.G.##
#B.$B.G#
#...G..#
########
```

Fig. 3: The default Sokoban level

3.1 Algorithm Analysis

As show in Algorithm 1, SARSA iterates over episodes and the steps in the episodes, updating the weights as it iterates. A major benefit to SARSA lies in the power of gradient descent and the weights the algorithm employs as it converges on an optimal value function. The importance of the weights lies in how, by updating the weights over time, each episode in the training process can project influence over many generations, as opposed to TD(0) learning which only takes observations at the current episode and the next episode.

As an on-policy method, SARSA commits to always "exploring" and trying to find the optimal policy that explores. However, as an on-policy method, it can converge to a local, not global, minima. On-policy methods are generally faster than off-policy methods, but aren't privy to the same level of flexibility. The "policy" SARSA is following is an ϵ-greedy one.

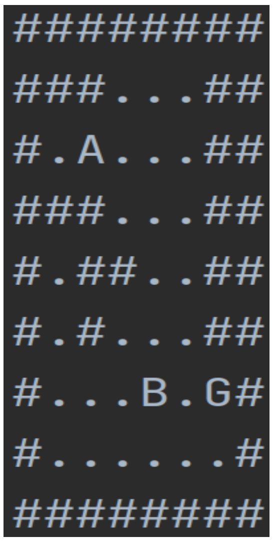
4 Experiments

When experimenting with SARSA, a modified version of the level displayed in Fig 3 was used, visible in Fig 4. This level contains only one box already close to the goal, meaning the agent should not have to follow some elaborate trajectories in order to reach the goal state. When experimenting, 2 different ones were run; a experiment only changing α and an experiment only changing ϵ. All were executed on the level with one box (Fig 4).

Keeping α set to a value of 0.1, γ set to 0.9,

Algorithm 1 SARSA Algorithm Pseudocode

```
for Each episode do
    S is the initial state of the episode
    A is the initial action of the episode (i.e. $\epsilon$-greedy)
    for Each step in the episode do
        Take action A, observe R, S'
        if S' is terminal then $w = w + \alpha[R - \hat{\boldsymbol{q}}(S, A, w)]\nabla\hat{\boldsymbol{q}}(S, A, w)$
            Go to next episode
        end if
        Choose A' as a function of $\hat{\boldsymbol{q}}(S, A, w)(i.e.\epsilon$-greedy)
      $w = w + \alpha[R + \gamma\hat{\boldsymbol{q}}(S', A', w) - \hat{\boldsymbol{q}}(S, A, w)]\nabla\hat{\boldsymbol{q}}(S, A, w)$
      S $\leftarrow S'$
      A $\leftarrow A'$
  end for
end for
```



```
########
###...##
#.A...##
###...##
#.##..##
#.#...##
#...B.G#
#......#
########
```

Fig. 4: A simplified default level

and the number of episodes to 100000 (with each episode performing at most 100 actions), the value of ϵ changed from 0.1 to 0.4 to 0.7, where the value of ϵ represents the probability that a greedy action will be chosen (i.e $\epsilon = 10 \rightarrow 10\%$ chance to select the greedy action). Further testing was done on the current state vector by subjecting it to three further experiments, with α ranging from 0.01 to 0.15 to 0.05, whilst ϵ remained static at 0.7.

5 Results

As can be seen in Figs 5, 6, and 7, the agent can find a value function that leads to the goal state, but it seems to only be able to achieve that when mostly choosing actions at random. It should be noted that Figs 5, 6, and 7 were all generated when utilizing the old "integer" representation of the state, and with ϵ represented by the values of 0.1 to 0.5 to 0.9.

Figs 8, 9, and 10 show the returned reward after 100000 episodes using the current state vector outlined in Section 3. As can be seen, the reward is (almost) identical to their "old state vector" counterparts, all returning an average reward of -0.001 per time step. Why could this be? Does it mean the old representation is equivalent to the current representation?

Not necessarily. All this means is that, within the 100000 episodes, each agent never reached the goal state, which is OK! The Sokoban puzzle is NP-Hard[2] and could take thousands upon millions of iterations of execution before it can consistently solve the problem. It is also important to recognize how the inherent randomness of the algorithm can affect results. Fig 11 was a second run of SARSA with $\alpha = 0.1$ and $\epsilon = 0.7$, but as can be seen the agent clearly reached the goal state fairly early on in training, but never reached it again.

The effect of α on the SARSA algorithm's training can be seen in Figs 12, 13, and 14. And whilst ϵ is responsible for greedy action selection, α represents the learning rate of the algorithm, the rate at which old information is replaced by new information. A factor of 0 means the agent learns nothing, whereas a factor of 1 allows the agent to only consider the most recent action.

5.1 Manhattan Distance

A last-minute addition to the Sokoban implementation, the Manhattan distance can be calculated using the following equation:

$$distance = |x_1 - x_2| + |y_1 - y_2|$$

where x and y are pairs of coordinates. This distance metric is used when the agent moves a box to determine if the box is moved closer to, or further away from, the closest goal tile. The intuition behind this additional reward is that when an agent moves a box closer to a goal, it is making progress towards the goal state, and such progress should be rewarded. A crucial component of this reward calculation is determining how far from the nearest goal a box actually is. Using euclidean distance will not suffice, as walls exist in the Sokoban levels and are not considered.

After implementing the Manhattan distance equation, the environment would, whenever a box was moved, find the closest goal tile (using the Manhattan distance), move the box, and check if the distance decreased or not. If the distance decreased, the agent was rewarded with 3 points, wheres they were penalized with -0.5 for *not* moving a box closer. Figs 15 and 16 illustrate the significant impact this metric had on training. The reward over episodes changes into a much more dynamic plot, an an interesting observation is that when the learning rate was increased, the obtained reward actually decreased after approximately 6500 to 7000 episodes as it had "forgotten" how to achieve the previous rewards.

5.2 More Boxes

In an additional experiment, the default level was loaded and the agent attempted to learn how to navigate it. with six boxes (as opposed to the one in the modified level), the agent would be required to learn how to navigate a much more complex level. As can be seen in Fig 17, the agent appears to have "forgotten" what it had learnt around episodes 9000. 10000, 15000, 21000, 24000, and 30000, but how is that possible?

Well, it isn't. The agent does not "forget" the weights so much as the weights are too large or small to be significant. While executing the 100000 episodes, the weight vector often displayed itself as a collection of `nan` values, indicating the values within the vector had reached positive or negative infinity. To remedy this, a more efficient state vector may be required. I would like to test a state vector that finds the centroid of all the boxes, and use the coordinates in place of all the coodinates of each individual box.

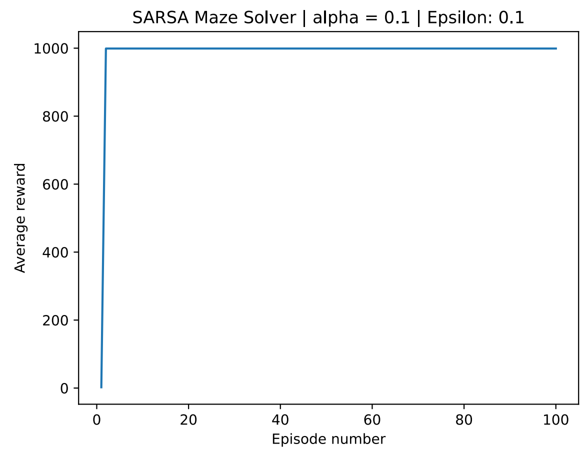


Fig. 5: A training instance where the agent reached the goal state consistently

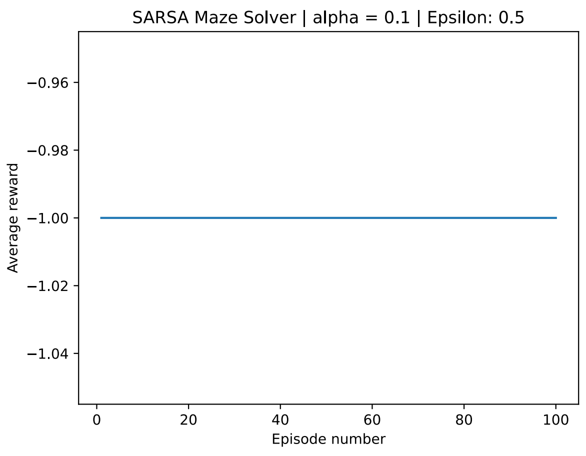


Fig. 6: A training instance where the goal state was never reached

5.2.1 Centroids

Fig 18 depicts the reward over time as SARSA employs the centroid state vector outlined in Section 5.2. The agent no longer "forgets" the weights and attempts to relearn what it had already spent episodes learning. It can be inferred that the weights are no longer diverging to infinity (positive *or* negative), allowing the weights to converge and the agent to learn!!

6 Conclusion

While SARSA is a more sophisticated reinforcement learning algorithm than TD(0), incorporating a weight vector and allowing that to become the policy for the MDP does not guarantee a better performance om the Sokoban problem. The ad-

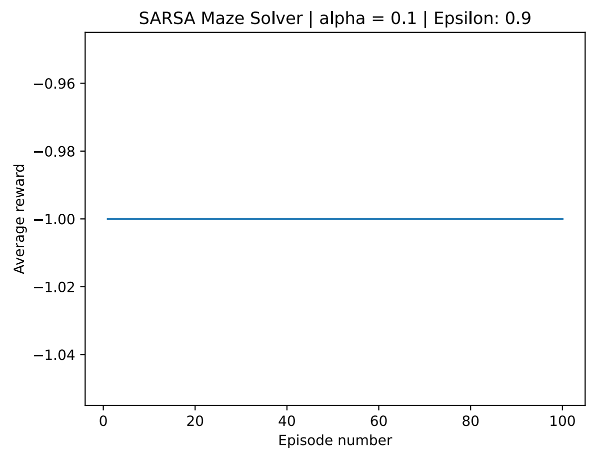


Fig. 7: Another goal state where the goal state was never reached

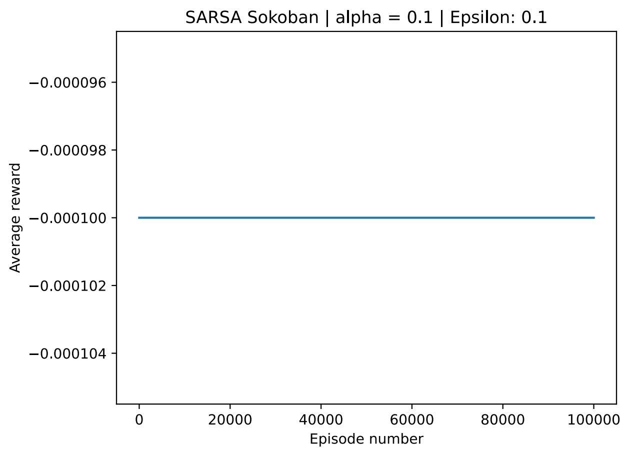


Fig. 8: Updated state vector, with α = 0.1 and ϵ = 0.1

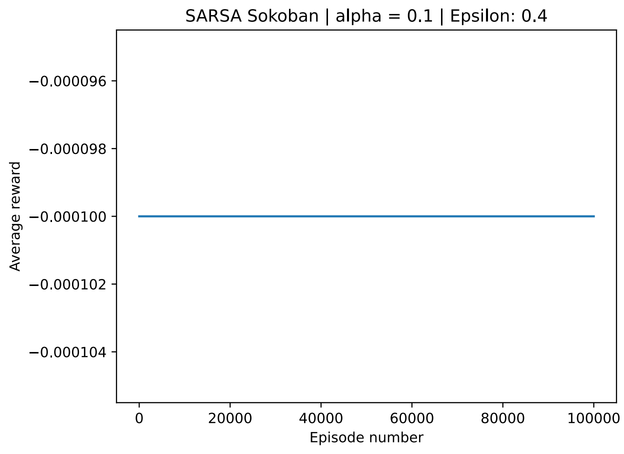


Fig. 9: Updated state vector, with α = 0.1 and ϵ = 0.4

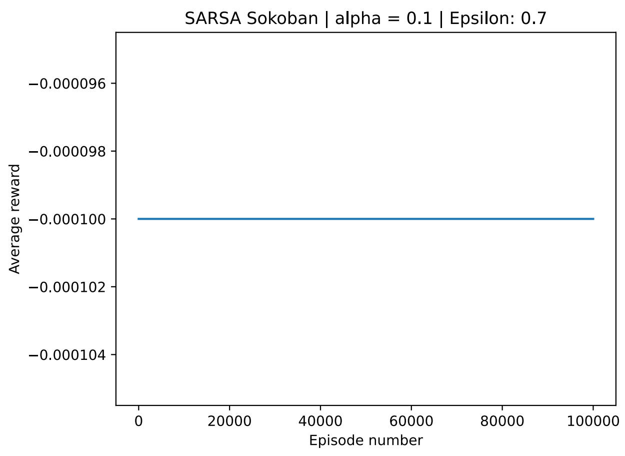


Fig. 10: Updated state vector, with α = 0.1 and ϵ = 0.7

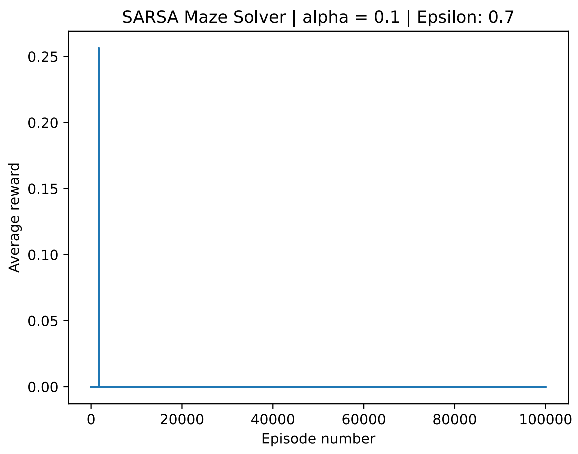


Fig. 11: Updated state vector, with α = 0.1 and ϵ = 0.7, run a second time

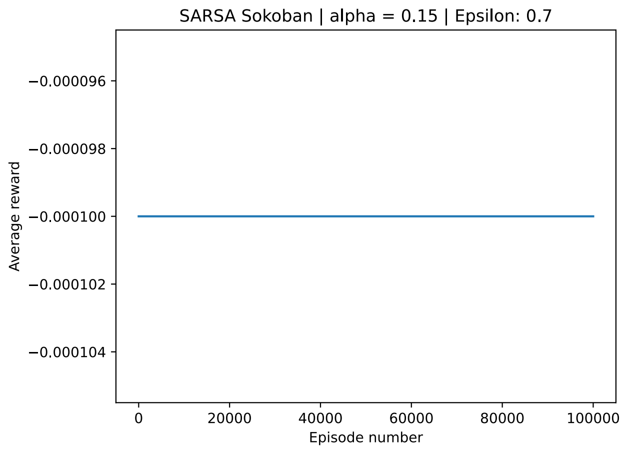


Fig. 12: Updated state vector, with α = 0.15 and ϵ = 0.7

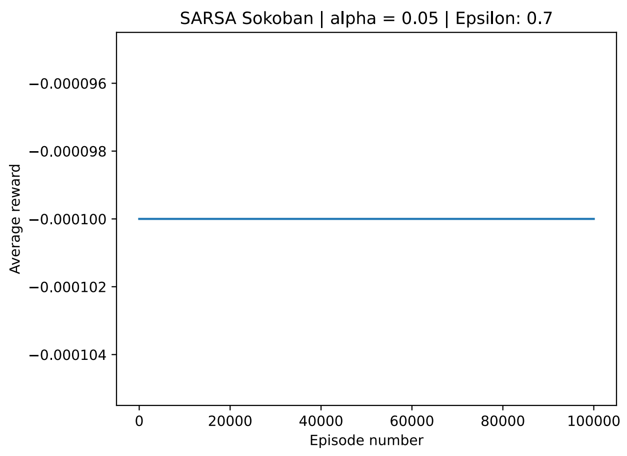


Fig. 13: Updated state vector, with $\alpha = 0.05$ and $\epsilon = 0.7$

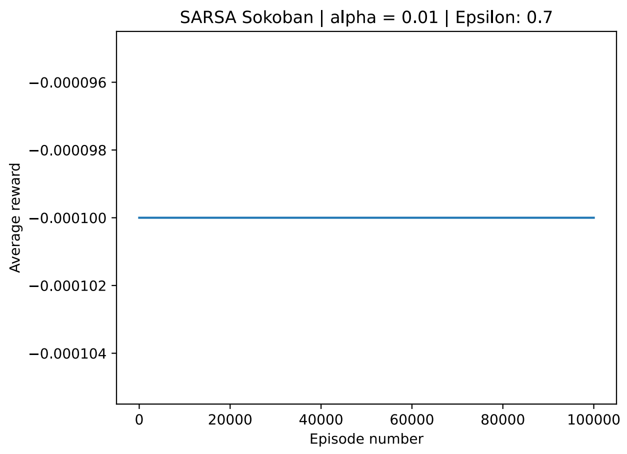


Fig. 14: Updated state vector, with $\alpha = 0.01$ and $\epsilon = 0.7$

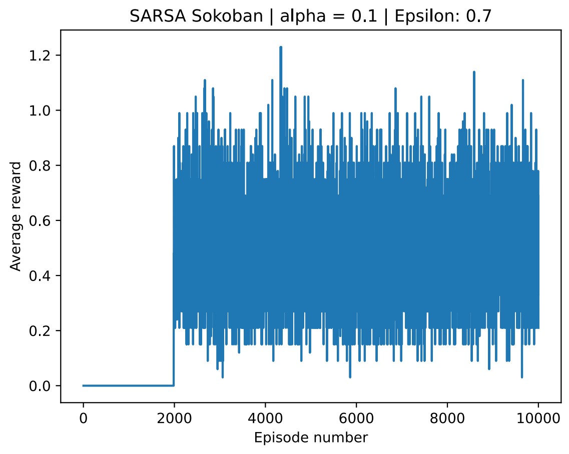


Fig. 15: Manhattan distance learning with $\alpha = 0.1$ and $\epsilon = 0.7$

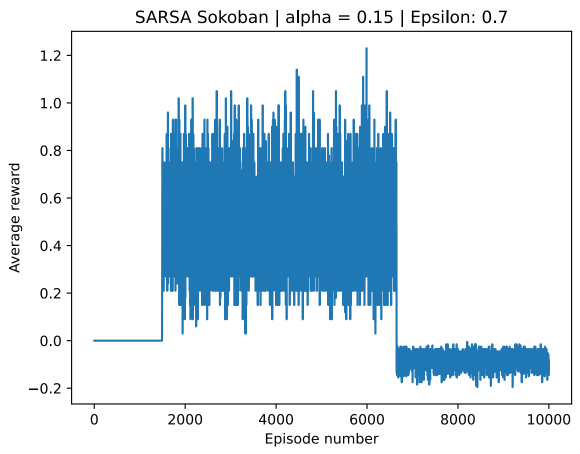


Fig. 16: Manhattan distance learning with $\alpha = 0.15$ and $\epsilon = 0.7$

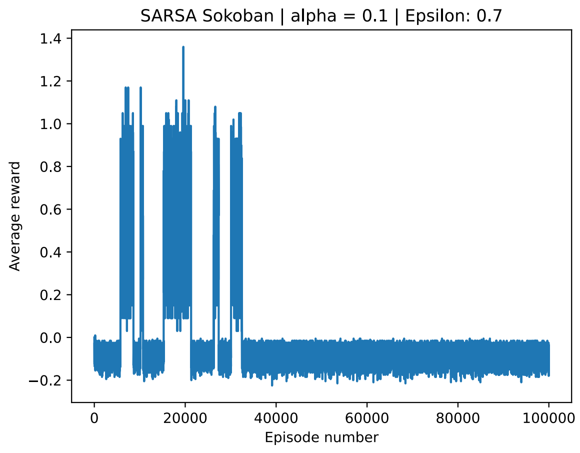


Fig. 17: Applying Manhattan distance to the default six-box level

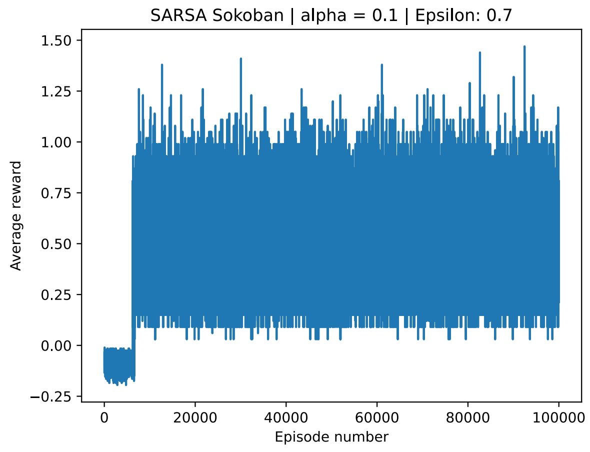


Fig. 18: Applying Manhattan distance to the default six-box level, utilizing the centroid state vector

dition that allowed the model to somewhat consistently gain a reward is the Manhattan distance measure, only applied when the agent moves a box. Utilizing the centroid of all box locations allowed the SARSA weights to converge and continuously learn. In future, it may be beneficial to rework the state vector and determine alternate state representations, as well as investigating alternative reward calculation techniques when an agent moves a box.

References

[1] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[2] Dorit Dor and Uri Zwick. Sokoban and other motion planning problems. *Computational Geometry*, 13(4):215–228, 1999.

[3] Martin L Puterman. Markov decision processes. *Handbooks in operations research and management science*, 2:331–434, 1990.

[4] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.