



UNIVERSITEIT•STELLENBOSCH•UNIVERSITY  
jou kennisvennoot • your knowledge partner

# **Learning to solve Sliding Puzzles using Reinforcement Learning**

Umr Barends  
18199313

Report submitted in partial fulfillment of the requirements of the module  
Project (E) 448 for the degree Baccalaureus in Engineering in the Department of  
Electrical and Electronic Engineering at Stellenbosch University.

Supervisor: JC Schoeman

November 8, 2020

# Acknowledgements

I would like to thank my supervisor Mr JC Schoeman for his excellent guidance and feedback, without which I would not have been able to complete this project. I would also to thank my family for their constant support and willingness to help me in whichever way possible. Finally I would also like to thank my pet cat Milo, for always being there when I needed him and reminding me that I needed to feed him.



UNIVERSITEIT • STELLENBOSCH • UNIVERSITY  
jou kennisvennoot • your knowledge partner

## Plagiaatverklaring / *Plagiarism Declaration*

1. Plagiaat is die oorneem en gebruik van die idees, materiaal en ander intellektuele eiendom van ander persone asof dit jou eie werk is.

*Plagiarism is the use of ideas, material and other intellectual property of another's work and to present is as my own.*

2. Ek erken dat die pleeg van plagiaat 'n strafbare oortreding is aangesien dit 'n vorm van diefstal is.

*I agree that plagiarism is a punishable offence because it constitutes theft.*

3. Ek verstaan ook dat direkte vertalings plagiaat is.


*I also understand that direct translations are plagiarism.*

4. Dienooreenkomstig is alle aanhalings en bydraes vanuit enige bron (ingesluit die internet) volledig verwys (erken). Ek erken dat die woordelike aanhaal van teks sonder aanhalingstekens (selfs al word die bron volledig erken) plagiaat is.

*Accordingly all quotations and contributions from any source whatsoever (including the internet) have been cited fully. I understand that the reproduction of text without quotation marks (even when the source is cited) is plagiarism*

5. Ek verklaar dat die werk in hierdie skryfstuk vervat, behalwe waar anders aangedui, my eie oorspronklike werk is en dat ek dit nie vantevore in die geheel of gedeeltelik ingehandig het vir bepunting in hierdie module/werkstuk of 'n ander module/werkstuk nie.

*I declare that the work contained in this assignment, except where otherwise stated, is my original work and that I have not previously (in its entirety or in part) submitted it for grading in this module/assignment or another module/assignment.*

Studentenommer / 18199313	Handtekening: 
Initials and surname / U.Barends	Datum / November 8, 2020

# Abstract

## English

In robotics, a manipulator (i.e. a robotic arm) is a device used to manipulate objects in its environment. When the manipulation task is relatively complex, it is often difficult or even impossible for a human to direct how the robot should act. One possible solution is that the robot learn by itself which actions are best, which can be done through reinforcement learning. A popular method for developing reinforcement learning algorithms for robotics is to first solve similar, less complex problems. In this project reinforcement learning is used to solve sliding puzzles. To accomplish this we use a specific method inspired by the way humans solve the puzzle, which makes it easier to apply reinforcement learning to solve the sliding puzzles.

## Afrikaans

In robotika is 'n manipuleerder (dit wil sê 'n robotarm) 'n toestel wat gebruik word om voorwerpe in sy omgewing te manipuleer. Wanneer die manipulasietask relatief ingewikkeld is, is dit vir 'n mens dikwels moeilik of selfs onmoontlik om te bepaal hoe die robot moet optree. Een moontlike oplossing is dat die robot op sigself leer watter aksies die beste is, wat gedoen kan word deur versterkingsleer. 'N Gewilde metode vir die ontwikkeling van versterkingsleeralgoritmes vir robotika is om eers soortgelyke, minder komplekse probleme op te los. In hierdie projek word versterkingsleer gebruik om glyraaisels op te los. Om dit te bewerkstellig, gebruik ons 'n spesifieke metode wat geïnspireer is deur die manier waarop mense die legkaart oplos, wat dit makliker maak om versterkingsleer toe te pas om die skuifraaisels op te los.

# Contents

<b>Declaration</b>	<b>ii</b>
<b>Abstract</b>	<b>iii</b>
<b>List of Figures</b>	<b>vi</b>
<b>Nomenclature</b>	<b>viii</b>
<b>1. Introduction</b>	<b>1</b>
1.1. Background . . . . .	1
1.2. Problem Statement and Project Objectives . . . . .	2
1.3. Document Outline . . . . .	2
<b>2. Literature Review</b>	<b>3</b>
2.1. Related Work . . . . .	3
2.2. Solvability of a Sliding Puzzle . . . . .	4
<b>3. Markov Decision Processes</b>	<b>6</b>
3.1. Definition of a MDP . . . . .	6
3.2. Value Function and Policies . . . . .	7
3.3. Bellman Expectation Equation . . . . .	8
3.4. Optimal Policies and Bellman Optimality Equations . . . . .	9
<b>4. Dynamic Programming</b>	<b>10</b>
4.1. Policy Evaluation . . . . .	10
4.2. Policy Improvement . . . . .	11
4.3. Policy Iteration . . . . .	12
4.4. Value Iteration . . . . .	12
4.5. Grid-world Example . . . . .	13
<b>5. Reinforcement Learning</b>	<b>16</b>
5.1. Model-free Prediction . . . . .	16
5.1.1. Monte Carlo Prediction . . . . .	16
5.1.2. Temporal Difference Learning . . . . .	18
5.2. Model-free Control . . . . .	20
5.2.1. Exploration and Exploitation . . . . .	21

5.2.2. SARSA: On-policy TD Control . . . . .	21
5.2.3. Q-learning: Off-policy TD Control . . . . .	22
<b>6. Applying Reinforcement Learning to Sliding Puzzles</b>	<b>24</b>
6.1. Representing 2x2 Puzzles . . . . .	24
6.1.1. Hyper-parameter Selection for 2x2 Sliding Puzzle . . . . .	25
6.2. Representing 3x3 Puzzles . . . . .	28
6.3. Sequential Approach for 3x3 Puzzles . . . . .	29
6.4. Experiments and Results . . . . .	31
<b>7. Summary and Conclusion</b>	<b>34</b>
<b>Bibliography</b>	<b>35</b>
<b>A. Project Planning Schedule</b>	<b>36</b>
<b>B. Outcomes Compliance</b>	<b>37</b>
<b>C. Student and Supervisor agreement</b>	<b>39</b>

# List of Figures

1.1.	Figure 1.1a depicts an arbitrary 3x3 sliding puzzle initial state and Figure 1.1b shows the solved puzzle. The puzzle is highlighted in different sections, to indicate how we will later in Chapter 6 break the puzzle into sub-problems. For example the tiles in red will be placed into its final positions first and the blue will be solved into place last. . . . .	2
2.1.	A representation showing the figure of a graph tree . . . . .	3
2.2.	Example of a sliding puzzle . . . . .	4
4.1.	Backup diagram inspired by Sutton and Barto [1]. . . . .	11
4.2.	Greedy policy iteration image inspired by Sutton and Barto [1]. The blue lines represents policy improvement and the red lines policy evaluation. By continuously using these two steps in succession, policy iteration eventually converge to the optimal state-value function $v_*$ and optimal policy $\pi_*$ . . . .	12
4.3.	Policy iteration, where we stop the iteration before the policy $\pi$ converges to the optimal policy $\pi^*$ . We can see this by the blue and red arrows stopping before reaching the point of convergence. . . . .	13
4.4.	This is a grid world example solved using value iteration. The goal is to have values in the cells represent the amount of steps it would take from that cell to the cell in the top left corner of that iteration. Each step results in a reward of -1. All iterations after iteration $k = 7$ are the same, although not shown, as the optimal state-value function $v_7(s) = v_*(s)$ has been reached by then. . . . .	14
4.5.	The arrows shows the optimal policy ( $k=7$ ) from Figure 4.4. The arrow heads indicates which directions have the least amount of cells in between the arrow containing and terminal (indicated by the red zero) cell. . . . .	14
6.1.	Sequence of 2x2 sliding puzzle states solved from an arbitrary starting state seen in 6.1a. The goal of the game is to arrange the tiles in ascending order read row by row, left to right with the blank tile in the bottom right corner as seen in 6.1d. The rules are described in detail in Chapter 1. . . . .	25

6.2.	Learning curve for agent training using SARSA for different values of $\epsilon$ , we keep constant $\gamma = 0.9$ and $\alpha = 0.1$ . The plot shows the total reward in an episode, averaged over 10000 samples. By sample, we refer to running the algorithm once and retrieving the total reward in each episode. We choose $\alpha$ and $\gamma$ arbitrarily.	26
6.3.	2x2 puzzle SARSA learning curve. $\gamma$ is varied while $\alpha = 0.1$ and $\epsilon = 0$	26
6.4.	2x2 puzzle SARSA learning curve. $\alpha$ is varied while $\gamma = 1$ and $\epsilon = 0$	27
6.5.	This plot shows the number of moves the agent takes to solve the 2x2 puzzle each episode. As the agent trains, we can see that it takes less moves to solve the puzzle. The agent was trained using SARSA with $\epsilon = 0$ , $\gamma = 1$ and $\alpha = 0.1$ .	28
6.6.	Puzzle size that Agent 1 sees and trains to solve. The tiles with X's can have any value from 2-8	29
6.7.	Puzzle size that Agent 2 sees and trains to solve. The puzzle in this figure represent the last two rows of the 3x3 puzzle.	30
6.8.	Puzzle size that Agent 3 sees and trains to solve. The puzzle in this figure represent the last two columns of the 3x3 puzzle.	30
6.9.	Puzzle size that Agent 4 sees and trains to solve. The puzzle in this figure represent the last two rows of the 3x3 puzzle.	30
6.10.	Puzzle that Agent 5 sees and trains on. The puzzle in this figure represents the bottom right grid of tiles in the 3x3 puzzle.	30
6.11.	SARSA vs Q-learning for Agent 1.	32
6.12.	The amount of moves that the combined 3x3 agent takes to solve the puzzle. The results are taken from the beginning to the end of the SARSA training.	32
6.13.	The histogram which shows the counts of the amount of moves that the combined 3x3 agent takes to solve the puzzle, while the agent is trained suing SARSA.	33
B.1.		37
B.2.		38
C.1.	Student and Supervisor agreement	39



# Nomenclature

## Variables and functions

$s, s'$	states
$a$	an action
$S$	set of all states
$R$	reward function
$t$	discrete time step
$G_t$	return after time t
$S_t$	state at time t
$R_t$	estimate of the expected reward at time given $\pi_t$
$P_{ss'^a}$	matrix of probabilities which predicts next state s'
$R_s^a$	the immediate reward after transitioning from state s to s' using action a
$\gamma$	discount factor applied to reward
$\pi$	policy, decision making rule
$\pi(s)$	action taken in state s using deterministic policy $\pi$
$\pi(a s)$	probability of taking action a given the state s using stochastic policy $\pi$
$v_\pi(s)$	value function of state s following policy $\pi$ , also known as the expected return
$v_*(s)$	value function of state s following the optimal policy
$q_\pi(s, a)$	value function of taking action a in state s following policy $\pi$
$q_*(s, a)$	value of taking action a in state s following the optimal policy
$\epsilon$	Probability of taking a random action using a greedy policy
$\alpha$	learning rate

## **Acronyms and abbreviations**

RL	Reinforcement Learning
SARSA	State-Action-Reward-State-Action
Q-learning	Q-value learning
MDP	Markov Decision Process
DP	Dynamic programming
TD learning	Temporal difference learning

# Chapter 1

## Introduction

In robotics, a manipulator (i.e. a robotic arm) is a device used to manipulate objects in its environment. When the manipulation task is relatively complex, it is often difficult or even impossible for a human to direct how the robot should act. One possible solution is that the robot learn by itself which actions are best, which can be done through reinforcement learning. A popular method for developing reinforcement learning algorithms for robotics is to first solve similar, less complex problems. In this project reinforcement learning is used to solve sliding puzzles. An example of a sliding puzzle being solved can be seen in Figure 1.1.

### 1.1. Background

The sliding puzzle is a game with a history dating back to the 1870's (Woolsey) [3]. The game consists of  $N^2 - 1$  tiles arranged on an  $N \times N$  grid with one empty tile. The tiles are numbered 1 to  $N$ . A possible configuration of a  $3 \times 3$  puzzle is shown in Figure 1.1a. The puzzle's state can be changed by sliding one of the numbered tiles, next to the empty tile, into the place of the empty tile. The empty tile then takes the place of the numbered tile.

The rules of the game are that the only tiles that can move are the ones next to the empty tile with no number, referred to from here on as the 'blank tile'. Additionally only one tile can be moved at a time. The final puzzle configuration is shown in 1.1b, where all the tiles are placed in ascending order read left to right and top to bottom. The aim of the game is to arrange a shuffled puzzle into the final puzzle configuration by moving one tile at a time. In this project, we take an arbitrary starting puzzle configuration, then solve it by using reinforcement learning.

Reinforcement learning is a sub-field of machine learning. According to Geron in [4], machine learning is the science of programming computers so it can learn from data. Reinforcement learning learns without being explicitly programmed. An example of the recent success of reinforcement learning, is Google's DeepMind AlphaGo program which beat the world champion of the game of Go. This is considered a breakthrough in the application of artificial intelligence.

1	2	3
5	7	6
4		8

(a)

1	2	3
4	5	6
7	8	

(b)

**Figure 1.1:** Figure 1.1a depicts an arbitrary 3x3 sliding puzzle initial state and Figure 1.1b shows the solved puzzle. The puzzle is highlighted in different sections, to indicate how we will later in Chapter 6 break the puzzle into sub-problems. For example the tiles in red will be placed into its final positions first and the blue will be solved into place last.

## 1.2. Problem Statement and Project Objectives

The aim of this project is to solve a shuffled sliding puzzle with the minimum amount of moves using reinforcement learning. The purpose of doing so is to later scale up the training method used, so that it can be applied to make a robot perform specific actions as per instruction. The objectives of this project is to produce a solution that trains the reinforcement learning agent in a way that is fast, robust, simple and scalable.

## 1.3. Document Outline

In Chapter 2, we will look at alternate approaches that can be used to solve sliding puzzles. These methods include various forms of search algorithms to solve the puzzles. In Chapter 3, we describe the theory relating to Markov decision processes, which forms the basis for modeling reinforcement learning problems.

In Chapter 4, we look at how dynamic programming can be used to solve problems, specifically problems that can be modeled using Markov decision processes. Chapter 5 discusses in detail the reinforcement learning techniques which we will use to solve our puzzle problem.

In Chapter 6, the theory of reinforcement learning is applied to solve sliding puzzles. We do so by first solving the 2x2 puzzle in order to make it easier for us to find the solution to solving the 3x3 puzzle.

In Chapter 7, we refer back to all the theory that we covered in the report and briefly discuss our results from Chapter 6.

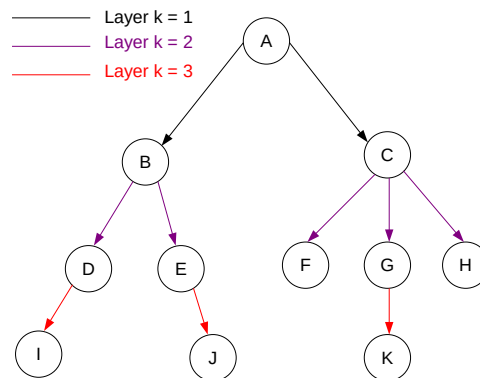
# Chapter 2

## Literature Review

In this chapter we will look at alternate methods to reinforcement learning that can be used to solve sliding puzzles. These are all methods that utilize search algorithms. We look at other research work, in order to avoid duplicating work already done on the topic of solving sliding puzzles.

### 2.1. Related Work

A 5x5 sliding puzzle has been solved using breadth first search by Korf in [6]. They solved the puzzle in a minimum of 2 hours and a maximum of over a month. The average amount of moves taken to solve the puzzle was 102.4 moves (Korf) [6]. Breadth first search (BFS) is a searching algorithm for traversing a graph tree. A graph consists of nodes and vertexes, in simpler terms a node can be thought of as a point and vertexes as the line connecting these points.



**Figure 2.1:** A representation showing the figure of a graph tree

An example graph can be seen in Figure 2.1, where the nodes are the circles, with each node containing a character as a value. The vertexes are colored for easy identification of the different layers in the graph. BFS works by searching through an entire layer at a time before moving to search the next layer. As BFS searches through all the nodes, it is

defined as being an exhaustive search method. Doing an exhaustive search to solve the sliding puzzle is not memory efficient, as the entire state space needs to be traversed.

Culberson [7], uses a variant of BFS, called IDA\*, where they use a method in which they store all states that are at most  $N$  moves of the blank tile away from the solved state in what they call an 'endgame database'. When the search arrives at a state in this set, the total amount of search steps taken is retrieved and the search is then stopped. Culberson [7] used  $N=8$  to reduce the total number of nodes traversed by 1000 fold. End game databases have also been used to solve Chess and Checkers games (Spaans) [8].

We now move to describing the background theory necessary to understand reinforcement learning theory, starting with the topic of Markov decision processes which will be explained in the next chapter.

## 2.2. Solvability of a Sliding Puzzle

Let us assume that we have an  $N \times N$  puzzle. We can represent the puzzle as an  $N \times N$  array, where we stack the array as a column vector  $1 \times N^2$ . For example see the  $4 \times 4$  puzzle in Figure 2.2 we have a  $1 \times 16$  array which we read row by row left to right as:

$$Array = (12, 7, 8, 13, 4, 9, 2, 11, 3, 6, 15, 14, 5, 1, 10).$$

In order to describe the conditions for a sliding puzzle to be solvable, we first define the term "inversion". We assume that the first index of the  $1 \times N^2$  array starts at the left top corner of a puzzle and that it runs from  $[0, (N \times N) - 1]$ . If we start looping through the values of the vector representation of the puzzle, then an inversion occurs when  $Array[index] > Array[index + 1]$ , where index is an arbitrary integer between 0 and  $N^2 - 1$ . Referring to Figure 2.2 we have a puzzle with a sum of inversions

$$\#inversions = 11 + 6 + 6 + 8 + 3 + 5 + 1 + 5 + 1 + 2 + 4 + 3 + 1 + 0 = 56.$$

12	7	8	13
4	9	2	11
3	6	15	14
5	1		10

**Figure 2.2:** Example of a sliding puzzle

According to Woolsey [12] for odd sized puzzles where  $N$  is odd we have the puzzle only being solvable if and only if the puzzle has an even number of inversions. For even

sized puzzles where  $N$  is even we have the board solvable if and only if the number of inversions plus the row of the blank square is odd. Hordern in [5], shows that for an  $N \times N$  sliding puzzle, exactly half of all possible puzzle configurations are solvable. By solvable, we mean that by sliding the blank piece in accordance with the rules, we can bring the puzzle into its solved configuration.

# Chapter 3

## Markov Decision Processes

In order to describe reinforcement learning techniques, we first have to find a way of mathematically defining reinforcement learning problems. In this chapter, we do so by formally introducing the concept of Markov decision processes (MDPs). Furthermore introducing the key concepts required to address the reinforcement learning problem. These concepts include; state and action space, returns, value functions and Bellman equations. The majority of the theory discussed in this chapter is based on the work by Sutton and Barto [1].

### 3.1. Definition of a MDP

A Markov decision process is a stochastic, discrete, time controlled sequence of states, actions and rewards. In other words a MDP is a partially controlled and random process as stated by Sutton and Barto [1]. In general RL problems can be described using MDPs. Weng [9] states that, MDP's are described by the following: state-space  $S$ , action-space  $A$ , state transition probability  $P_{ss'}^a$ , reward function  $R_s^a$  and discount factor  $\gamma$ . These variables can be denoted in a 5-tuple as

$$(S, A, P_{ss'}^a, R_s^a, \gamma) \quad (3.1)$$

where,

- $S$  is a finite set of states with state  $s \in S$
- $A$  is a finite set of actions with action  $a \in A$
- $P_{ss'}^a$  is a matrix of probabilities which predicts the next state  $s'$ , where  $P_{ss'}^a = P(S_{t+1} = s' | S_t = s, A_t = a)$
- $R_s^a$  is the immediate reward after transitioning from state  $s$  to  $s'$  using action  $a$ , where  $R_s^a = E[R_{t+1} | S_t = s, A_t = a]$
- $\gamma \in [0,1]$  is the discount factor applied to the reward



In a MDP, the assumption is that all the states have the Markov property (Sutton and Barto) [1]. This means that

$$P[S_{t+1}|S_t] = P[S_{t+1}|S_1, \dots, S_t]. \quad (3.2)$$

In other words, the probability of transitioning to state  $S_{t+1}$  given state  $S_t$ , must be equal to the probability of transitioning to state  $S_{t+1}$  given all states  $S_1$  to  $S_t$ . This means that the current state is required to contain all the information of the previous states.

## 3.2. Value Function and Policies

The return  $G_t$  is the total sum of the reward at each time step, multiplied by a constant diminishing factor  $\gamma$ . The discount  $\gamma \in [0,1]$  determines the present value of future reward  $R_t$ .

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots \\ &= \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \end{aligned} \quad (3.3)$$

For  $\gamma = 0$ , the return  $G_t$  only depends on the single reward that can be obtained in the next step, namely  $R_{t+1}$ . When  $\gamma = 0$  the return  $G_t$  is 'short-sighted'. For  $\gamma = 1$  on the other hand, the return depends on all the rewards that are projected to be obtained until the process terminates.  $G_t$  is thus "far-sighted" for  $\gamma = 1$  as per Sutton and Barto [1]. What is also noticeable is that using a discount factor between 0 and 1 makes the return finite because  $R_{t+k+1}$  is finite and

$$\sum_{k=0}^{\infty} \gamma^k = \frac{1}{1 - \gamma}, \quad (3.4)$$

which is finite for  $\gamma \in [0,1]$ .

The probability that the agent will choose a specific action given a state is called its policy  $\pi(a|s)$ . Mathematically this is defined as

$$\pi(a|s) = P[A_t = a, S_t = s]. \quad (3.5)$$

Another definition is the value function  $v(s)$  defined as

$$v(s) = E[G_t | S_t = s]. \quad (3.6)$$

The value function  $v(s)$  is the expected return given that  $v(s)$  is evaluated in state  $s$  at

time-step  $t$ .

### 3.3. Bellman Expectation Equation

Sutton and Barto [1] decompose  $v(s)$  so that it becomes a recursive function as follows

$$v(s) = E[G_t | S_t = s] \quad (3.7)$$

$$= E[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots | S_t = s] \quad (3.8)$$

$$= E[R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \dots) | S_t = s] \quad (3.9)$$

$$= E[R_{t+1} + \gamma G_{t+1} | S_t = s] \quad (3.10)$$

$$= E[R_{t+1} + \gamma v(S_{t+1}) | S_t = s]. \quad (3.11)$$

This means that  $v(s)$  only depends on  $R_{t+1}$  (the reward that can be obtained in the next step) and  $\gamma v(S_{t+1})$ , the discounted value function at time  $t+1$ . What has been defined in Equation 3.11 is known as the Bellman equation as per Sutton and Barto [1]. Sutton and Barto [1] then define the value function given that the agent follows the policy  $\pi$  as:

$$v_\pi(s) = E_\pi[G_t | S_t = s]. \quad (3.12)$$

Additionally, they define the action-value function (the q-function),

$$q_\pi(s, a) = E_\pi[G_t | S_t = s, A_t = a]. \quad (3.13)$$

The q-function is defined as the expected return of taking action  $a$  from state  $s$  and thereafter following policy  $\pi$ . What can be noted is that  $v(s)$  is a prediction of what the value for being in a certain state is, which is related to planning. While  $q(s, a)$  is the value for being in a certain state and taking an action, which relates to control.

Once again, Sutton and Barto [1] decompose the state-value and action-value functions to be in the form of the Bellman Equation in equation 3.11. This results in the Bellman *expectation* equations:

$$v_\pi(s) = E_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s] \quad (3.14)$$

$$= \sum_{a' \in A} \pi(a' | s) (R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_\pi(s')) \quad (3.15)$$

and

$$q_\pi(s, a) = E_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a] \quad (3.16)$$

$$= R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \sum_{a' \in A} \pi(a' | s') q_\pi(s', a'). \quad (3.17)$$

Equation 3.15 can be placed into vector form as:

$$v_\pi = R^\pi + \gamma P^\pi v_\pi \quad (3.18)$$

Which has the solution:

$$v_\pi = (I - \gamma P^\pi)^{-1} R^\pi \quad (3.19)$$

Equation 3.19 is a concise formula which can be easily implemented in code.

### 3.4. Optimal Policies and Bellman Optimality Equations

Sutton and Barto [1] now define the optimal policy, which only has value 1 if the action the agent takes maximizes  $q_\pi(s, a)$ . This is mathematically defined as:

$$\pi_*(a|s) = \begin{cases} 1, & \text{if } a = \arg \max_{a \in A} q_\pi(s, a) \\ 0, & \text{otherwise} \end{cases} \quad (3.20)$$

There always exists an optimal deterministic policy for any MDP (Sutton and Barto) [1]. Sutton and Barto [1] then define what is known as the Bellman *optimality* equations.

$$v_*(s) = \max_a (R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_*(s')) \quad (3.21)$$

and

$$q_*(s, a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \max_{a'} q_*(s', a'). \quad (3.22)$$

We only need to know  $q_*(s, a)$  to know the optimal policy. This is important because the optimal policy describes which actions must be taken to maximize rewards, which is precisely the goal of RL. What equation 3.21 represents is the maximum predicted return that can be calculated from state  $s$ . Equation 3.22 represents the maximum return that can be obtained by first taking action  $a$  in state  $s$  and thereafter following the policy  $\pi_*$ . At this point, we have outlined a mathematical background for reinforcement learning problems. In the next two chapters we will discuss methods of solving the MDP and obtaining the optimal state-value and action-value functions.

# Chapter 4

## Dynamic Programming

Dynamic programming (DP) is a method to solve problems by using a divide and conquer approach as per Silver [10]. That is to say a problem is broken into smaller sub-problems and then solved. Dynamic programming is used to solve MDPs of which we have perfect models. Most times, we do not have access to such models and hence DP is of limited use in reinforcement learning. However, DP is still of theoretical importance and a useful component to understanding reinforcement learning as stated by Sutton and Barto [1]. There are two properties that must be met for a problem to be solvable via DP, namely optimal substructure and overlapping sub-problems as per Silver [10]. *Optimal substructure* requires that the solution can be decomposed into sub-problems. *Overlapping sub-problems* requires that the sub-problems recur, so that the solution can be cached and reused.

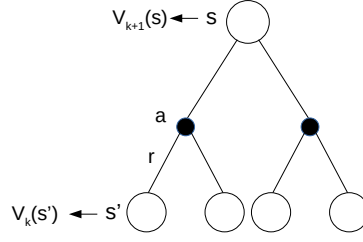
Markov decision processes satisfy both of these conditions. According to Silver [10], the Bellman equation provides a recursive decomposition which fulfills the optimal substructure property, while the value function caches and reuses solutions satisfying the overlapping sub-problem's condition. The key concept of DP is to use state-value functions as the structure through which optimal policies are found (Sutton and Barto) [1]. As a basis for finding the optimal policy, the next section first discusses how to evaluate a given policy.

### 4.1. Policy Evaluation

In order to evaluate a given policy, we calculate the state-value function  $V_\pi$  for a given policy  $\pi$ . Sutton and Barto [1] names this policy evaluation. From Equation 3.15 we know that,

$$v_\pi(s) = \sum_{a' \in A} \pi(a'|s) (R_s^{a'} + \gamma \sum_{s' \in S} P_{ss'}^{a'} v_\pi(s')).$$

Figure 4.1 depicts a sequence of states and actions and is known as a backup diagram. It is known as a backup diagram as it shows the transference of information from the future state  $s'$  to a present state  $s$ . We can explain how policy evaluation works at the hand of the backup diagram in Figure 4.1. Starting at the root node (state  $s$ ) the agent can take any action according to its policy  $\pi$ . After the agent takes this action, the environment responds and moves the agent to state  $s'$ . The environment's response depends on the probabilities contained in the function  $P_{ss'}^a$ .



**Figure 4.1:** Backup diagram inspired by Sutton and Barto [1].

Sutton and Barto [1] then introduce an approximate value function  $v_k(s)$  (indexed by  $k$ ) which maps every state  $s \in S$  to a real number in  $\mathfrak{R}$ . Then the Bellman optimality equations are used recursively in order to update  $v_k(s)$  to  $v_{k+1}(s)$  as can be seen in Figure 4.1. Sutton and Barto [1] then choose  $v_0(s)$  arbitrarily, where after  $v_k(s)$  is iteratively updated using

$$v_{k+1}(s) = \sum_{a' \in A} \pi(a'|s) (R_s^{a'} + \gamma \sum_{s' \in S} P_{ss'}^{a'} v_k(s')). \quad (4.1)$$

The value function is updated continuously in this way until convergence, when the difference between  $v_k(s)$  and  $v_{k+1}(s)$  is determined negligible. This process is known as iterative policy evaluation.

Now that we have a tool to evaluate a policy  $\pi$ , we will discuss how to improve a given policy in order to find the optimal policy  $\pi_*$ .

## 4.2. Policy Improvement

The purpose of policy improvement is to use the value function of a given policy to alter how the agent should choose actions. In order to improve the policy Sutton and Barto introduce a formula for computing a new greedy policy  $\pi'$  as,

$$\pi' = \text{greedy}(v_\pi), \quad (4.2)$$

$$\pi'(s) = \arg \max_a (R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_\pi(s')). \quad (4.3)$$

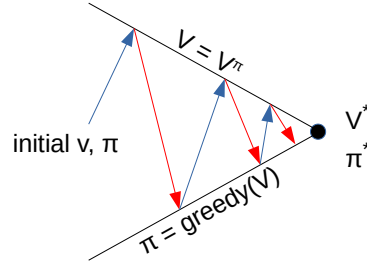
Sutton and Barto [1] proves that using Equation 4.3 will always result in a new policy at least as good as the old one. What it means for a policy to be better or at least as good as another is that the value function  $v_{\pi'}(s)$  must be equal or greater than  $v_\pi(s)$  for all states, mathematically presented as:

$$v_{\pi'}(s) \geq v_\pi(s) \quad \forall s. \quad (4.4)$$

In the next two sections we will combine policy evaluation and policy improvement as a tool to find the optimal policy via two methods, policy iteration and value iteration.

### 4.3. Policy Iteration

When a given policy  $\pi$  is improved using  $v_\pi$  to yield a better policy  $\pi'$ , we can then compute  $v_{\pi'}$  and improve it again resulting in an even better  $\pi''$ . By iteratively improving the value function and policy, we can obtain a monotonically improving sequence of policies and value functions. This procedure (greedy policy iteration) can be seen in Figure 4.2, where the blue line represents policy improvement and the red line policy evaluation.



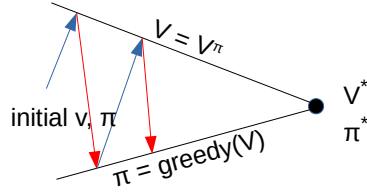
**Figure 4.2:** Greedy policy iteration image inspired by Sutton and Barto [1]. The blue lines represents policy improvement and the red lines policy evaluation. By continuously using these two steps in succession, policy iteration eventually converge to the optimal state-value function  $v_*$  and optimal policy  $\pi_*$ .

### 4.4. Value Iteration

Policy evaluation and policy improvement are iterative processes. Policy iteration loops over both of these methods requiring many calculations across the entire state-space (Sutton and Barto) [1]. Furthermore Iterative policy evaluation only converges in the limit. However Sutton and Barto [1] state that we can stop the iteration process before the policy converges to the optimal policy  $\pi_*$ . This can be seen in Figure 4.3. The special case of stopping policy iteration after one step, is called value iteration.

Value iteration can be mathematically described by:

$$v_{k+1}(s) = \max_{a \in A} (R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_k(s')) \quad (4.5)$$



**Figure 4.3:** Policy iteration, where we stop the iteration before the policy  $\pi$  converges to the optimal policy  $\pi^*$ . We can see this by the blue and red arrows stopping before reaching the point of convergence.

and in vector form,

$$\mathbf{v}_{k+1} = \max_a (\mathbf{R}^a + \gamma \mathbf{P}^a \mathbf{v}_k). \quad (4.6)$$

Note these equations are the Bellman optimality equation (in equation 3.21), turned into an update rule. Algorithm 1 from Sutton and Barto [1] shows how value iteration can be implemented.

---

**Algorithm 1:** Value Iteration, for estimating  $\pi \approx \pi_*$

---

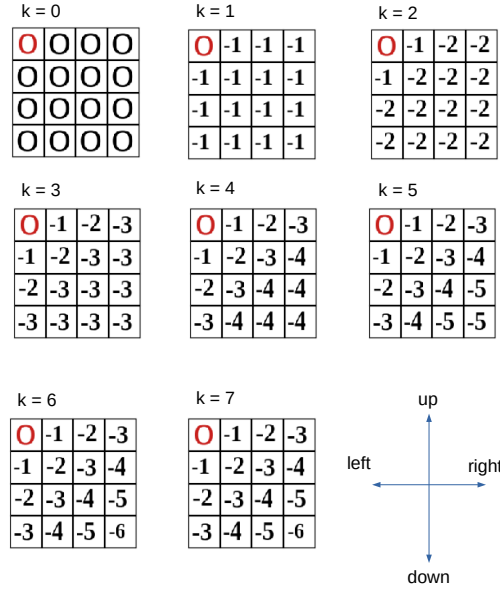
**Input** : A small threshold  $\theta > 0$  determining accuracy of estimation

**Output** : A deterministic policy  $\pi \approx \pi_*$

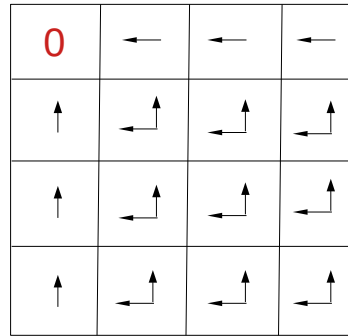
- 1: Initialize  $V(s)$ ,  $\forall s \in S$ , arbitrarily except that  $V(\text{terminal}) = 0$
  - 2: Loop:
  - 3:      $\delta \leftarrow 0$
  - 4:     Loop for each  $s \in S$ :
  - 5:          $v \leftarrow V(s)$
  - 6:          $V(s) \leftarrow \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma V(s')]$
  - 7:          $\delta \leftarrow \max(\delta, |v - V(s)|)$
  - 8:      $\delta < \theta$
  - 9:     Output a deterministic policy,  $\pi \approx \pi_*$ , such that
  - 10:          $\pi(s) = \arg \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma V(s')]$
- 

## 4.5. Grid-world Example

In this section we provide an example to aid the discussion of dynamic programming, specifically value iteration. Figure 4.4 shows an example problem, which Silver [10] uses to illustrate how value iteration works. The states are defined as cells contained within a grid. Figure 4.4 shows the results of value iteration being applied to find the shortest path



**Figure 4.4:** This is a grid world example solved using value iteration. The goal is to have values in the cells represent the amount of steps it would take from that cell to the cell in the top left corner of that iteration. Each step results in a reward of -1. All iterations after iteration  $k=7$  are the same, although not shown, as the optimal state-value function  $v_7(s) = v_*(s)$  has been reached by then.



**Figure 4.5:** The arrows shows the optimal policy (k=7) from Figure 4.4. The arrow heads indicates which directions have the least amount of cells in between the arrow containing and terminal (indicated by the red zero) cell.

between a cell in a 4x4 grid and the terminal cell, indicated by the red zero. The value function for  $k=0$  is initialized as  $v_0(s) = 0$ . The reward function is chosen as  $R = -1$  for all state-action pairs. The probability function is chosen as  $P(s', r|s, a) = 1$  and  $\gamma = 1$ . Using Algorithm 1 with these parameters, results in a value function converging in seven iterations as seen in Figure 4.4. The diagram in Figure 4.5 shows the optimal policy found using the computed optimal value function  $v_7(s)$ .



In the next chapter we use a similar method to value iteration to find the optimal policy, except that we do not assume that we have access to a perfect model of the MDP.

# Chapter 5

## Reinforcement Learning

In the previous chapter we discussed methods for solving decision-making problems which require knowledge about the model of the environment. This knowledge was represented in the probability transition matrix  $P$ . Often it is either not possible to know the model of the environment beforehand, or even if the environment is known it is too expensive to store or use. This is where reinforcement learning is useful because it does not require a model of the environment.

In reinforcement learning, an agent learns the best actions to take, by observing sequences of states and actions, called episodes. When the agent reaches the final state, we say the episode has terminated. In this chapter we discuss two methods of reinforcement learning, namely Monte Carlo and temporal difference (TD) learning, either of which can be used for solving prediction or control problems. In the next section, we discuss model-free prediction methods.

### 5.1. Model-free Prediction

Recall that for prediction we do not wish to optimise a policy, but rather just evaluate a given policy by finding the value function. In the next two subsections we discuss how this problem can be solved by either using Monte Carlo or TD learning. The aim of Monte Carlo learning is to find a value function  $v_\pi$  by averaging sample returns. Alternatively the aim of TD learning is to learn a value function  $v_\pi$  from experience by following policy  $\pi$ .

#### 5.1.1. Monte Carlo Prediction

According to Sutton and Barto [1], the term Monte Carlo is generally used for any estimation technique where a large portion of its computation is random. Sutton and Barto [1] specifically describe Monte Carlo methods as solving reinforcement learning problems by averaging sample returns. Monte-Carlo methods require only observations, which it obtains by sampling sequences of states, actions and rewards. By sampling enough times, by the law of large numbers, eventually an accurate state-value function will be obtained (Sutton and Barto) [1].

---

**Algorithm 2:** Monte Carlo prediction algorithm for estimating  $V \approx v_\pi$  (Sutton and Barto) [1]

---

**Input** : policy  $\pi$  to be evaluated

**Output** :  $V(S_t)$

```

1: Initialize:
2:    $V(s) \in \mathbb{R}$  arbitrarily  $\forall s \in S$ 
3:    $\text{returns}(s) \leftarrow$  empty list  $\forall s \in S$ 
4: loop infinitely for each episode:
5:   Generate an episode following  $\pi$ 
6:    $G \leftarrow 0$ 
7:   for each step in episode
8:      $G \leftarrow \gamma G + R_{t+1}$ 
9:     If  $S_t$  appears in  $S_0, S_1, \dots, S_t - 1$ :
10:      Append  $G$  to  $\text{returns}(S_t)$ 
11:       $V(S_t) \leftarrow \text{average}(\text{returns}(S_t))$ 

```

---

Algorithm 2 shows how Monte Carlo prediction can be implemented. Throughout the following discussion we will refer back to the algorithm. Monte-Carlo policy evaluation works as follows. We define a total return  $S$  in state  $s$ , seen in line 3 of Algorithm 2, as  $S(s)$  and a integer counter  $N(s)$  which represents the amount of times  $s$  has been seen. We then initialize both  $S(s)$  and  $N(s)$  initialized to zero, which corresponds to line 6 of Algorithm 2.

Then at time step  $t$  and state  $s$  we increment the counter  $N(s) = N(s) + 1$ . This can be done in two ways, one being only incrementing  $N(s)$  the *first* time state  $s$  is seen, the other being incrementing  $N(s)$  *every* time states  $s$  is seen (Sutton and Barto) [1]. In this chapter we use the method of incrementing the counter only the first instance that state  $s$  is seen. Then we increment the total reward  $S(s) = S(s) + G_t$  every time the agent reaches state  $s$ . This corresponds to line 8 of Algorithm 2.

Finally we calculate the state value estimate function as  $V(s) = \frac{S(s)}{N(s)}$ , which corresponds to line 11 of Algorithm 2. Then  $V(s) \rightarrow V_\pi(s)$  as  $N(s) \rightarrow \infty, \forall s \in S$ . In other words, when we have seen all states infinitely many times, we will have obtained exactly  $V_\pi(s)$ . Although theoretically we need to visit each state infinitely many times, practically there is a finite time-step where  $V(s)$  is close enough to  $V_\pi(s)$  that we terminate the process. In order to describe incremental Monte-Carlo updates, Silver [10] introduces the concept of an incremental mean  $u_k$ . We can further use the definition of expectation to express the

incremental mean recursively as

$$u_k = \frac{1}{k} \sum_{j=1}^k x_j \quad (5.1)$$

$$\begin{aligned} &= \frac{1}{k} \left( x_k + \sum_{j=1}^{k-1} x_j \right) \\ &= \frac{1}{k} (x_k + (k-1)u_{k-1}) \\ &= u_{k-1} + \frac{1}{k} (x_k - u_{k-1}). \end{aligned} \quad (5.2)$$

Since  $V(s)$  is a mean, we break it up similarly to Equation 5.2, with  $u_{k-1} = V(s)$ ,  $N(s) = k$  and return  $G_t = x_k$ . Substituting these variables into equation 5.2 yields that at every time-step  $t$ :

$$N(S_t) \leftarrow N(S_t) + 1 \quad (5.3)$$

$$V(S_t) \leftarrow V(S_t) + \frac{1}{N(S_t)} (G_t - V(S_t)). \quad (5.4)$$

Sutton and Barto in [1] sets  $\frac{1}{N(S_t)} = \alpha$ , where  $0 < \alpha < 1$  resulting in

$$V(S_t) \leftarrow V(S_t) + \alpha (G_t - V(S_t)). \quad (5.5)$$

The reason for using a constant  $\alpha$  instead of the coefficient  $\frac{1}{N(S_t)}$  that changes with time, is so that we can control how much the state-value function  $V(S_t)$  is updated towards the target  $[G_t - V(S_t)]$ . Equation 5.5 gives us a way to find the optimal state-value function  $V_\pi(S_t)$  by iteratively updating  $V(S_t)$ . Sutton and Barto names Equation 5.5, constant- $\alpha$  Monte Carlo.

### 5.1.2. Temporal Difference Learning

Temporal difference (TD) learning uses a combination of Monte Carlo and dynamic programming (DP) ideas. TD learning is unique to the field of reinforcement learning, unlike Monte Carlo and DP which have more general applications besides reinforcement learning (Sutton and Barto) [1]. Like Monte Carlo learning, TD learning can learn without requiring a model of the environment.

The motivation behind TD learning is that we do not use entire sequences, but rather use the value function of the next state as an estimate. This results in the benefit of not having to wait to the end of an episode. The TD target  $[R_{t+1} + \gamma V(S_{t+1})]$  is the value which  $V(S_t)$  tends to as  $t \rightarrow \infty$  and  $[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$  is known as the TD error. These two terms can be seen on line 8 of Algorithm 3. The aim of TD learning is to reduce

---

**Algorithm 3:** TD learning algorithm for estimating  $v_\pi$  (Sutton and Barto [1])

---

**Input** : policy  $\pi$  to be evaluated, step size  $\alpha \in (0, 1]$

**Output** :  $V(S)$

- 1: Initialize:  $V(s)$  arbitrarily  $\forall s \in S$
  - 2: Initialize:  $V(\text{terminal state}) = 0$
  - 3: for each episode:
  - 4:     Initialize  $S$
  - 5:     for each step in episode
  - 6:          $A \leftarrow$  action obtained from  $\pi$  for  $S$
  - 7:         Take action  $A$ , observe  $R, S'$
  - 8:          $V(s) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$
  - 9:          $S \leftarrow S'$
  - 10:     break when  $S$  is terminal
- 

the TD error towards zero and update  $V(S_t)$  to the estimated return  $R_{t+1} + \gamma V(S_{t+1})$ . The simplest form of TD is replacing the return  $G_t$  in Equation 5.5 which results in the equation that is on line 8 of Algorithm 3:

$$V(S_t) = V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)). \quad (5.6)$$

As TD updates the state value function  $V(s)$  towards an estimate, Sutton and Barto [1] call it a bootstrapping method. Recall from Chapter 3 that we can define the state value function under policy  $\pi$  as:

$$v_\pi(s) = E_\pi[G_t | S_t = s] \quad (5.7)$$

$$= E_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s] \quad (5.8)$$

$$= E_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s] \quad (5.9)$$

Sutton and Barto [1] state that the Monte Carlo target is an estimate as it uses a sample return in place of the expected value in Equation 5.7. They further also state that dynamic programming uses a target that is an estimate as it uses an approximation of  $v_\pi(S_{t+1})$ , namely  $V(S_{t+1})$ . Sutton and Barto [1] then reason that the TD target is an estimate because it uses both Monte Carlo and dynamic programming, as it samples the expected value in Equation 5.9 and it uses the current estimate  $V$  as an approximation for  $v_\pi$ . Therefore TD gains the benefit of sampling from Monte Carlo and the benefit of bootstrapping from dynamic programming. We now look at a more general view of the return  $G_t$ , in order to make clear the relationship between Monte Carlo and TD learning.

## Relationship Between Monte Carlo and TD Learning

Sutton and Barto [1] mathematically describe the relationship between MC and TD using what they define as the  $n$ -step return  $G_t^{(n)}$  described by:

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n}). \quad (5.10)$$

Although we do not implement the  $n$ -step return in this report, we show it to clarify the relationship between Monte Carlo and TD learning. Additionally, we show  $G_t^{(n)}$  for a few values of  $n$  as follows:

$$(TD)n = 1 : G_t^{(1)} = R_{t+1} + \gamma V(S_{t+1}) \quad (5.11)$$

$$n = 2 : G_t^{(2)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 V(S_{t+2}) \quad (5.12)$$

$$(MC)n = \infty : G_t^{(\infty)} = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{T-1} R_T. \quad (5.13)$$

What can be seen is, as  $n \rightarrow \infty$ , the return in TD learning turns into the same form as in Monte Carlo. Hence Monte Carlo can be said to be a special case of TD learning. Substituting  $G_t^{(n)}$  into Equation 5.6 yields what Sutton and Barto [1] calls  $n$ -step TD as:

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t^{(n)} - V(S_t)). \quad (5.14)$$

What we now have is a method to control how many steps we want an agent to look ahead. This is useful because  $V(S_t)$  can converge more quickly to  $V_\pi(S_t)$  if  $n$  is chosen well. Normally one has to test for different values of  $n$  to see which value works best for the specific problem at hand.

## 5.2. Model-free Control

We no longer look at estimating the state-value function  $V_\pi(S_t)$ , but instead work with *state-action* pairs in order to find the optimal action-value function  $q_*(s, a)$ . We do so to find the optimal policy  $\pi_*$  from  $q_*(s, a)$ . In order to find the optimal action-value function  $q_*(s, a)$ , we require a method of evaluating an action-value policy. Even if we have the optimal state value function, we still need the model to find the optimal policy. Previously, when using greedy policy improvement over the state-value function  $V(s)$  we had in Equation 4.2:

$$\pi'(s) = \arg \max_{a \in A} (R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_*(s')),$$

where it can be seen that we need the model  $P_{ss'}^a$  and the reward function  $R_s^a$ . Therefore Sutton and Barto [1] applied the greedy policy improvement over the state action function  $Q(s,a)$  as follows:

$$\pi'(s) = \arg \max_{a \in A} Q(s, a). \quad (5.15)$$

This equation is derived from the fact that  $q(s,a)$  gives us a comparison between actions that  $v(s)$  doesn't. One should notice that to find the optimal policy (by using Equation 5.15 for policy improvement), we no longer need a model of the environment. In other words there is no dependency on the probability matrix  $P_{ss'}^a$  nor the reward function  $R_s^a$ . In the next section, we will see that using Equation 5.12 exactly as is would run into a problem. This is due to the trade-off between exploration and exploitation. In the same section a solution is then given to the problem.

### 5.2.1. Exploration and Exploitation

In reinforcement learning, exploration refers to an agent taking a random action and exploitation when the agent acts greedily (Tokic) [11]. When an agent exploits it can obtain a larger return and when it explores it gets a more accurate estimate of the action-value function. The agent can not simultaneously explore and exploit which results in what is known as the exploration-exploitation dilemma (Tokic) [11]. To ensure that the agent explores occasionally instead of always taking the greedy action with respect to the action-value function  $Q(s,a)$ , we let the agent act randomly with a small probability  $\epsilon$ . This can mathematically be expressed as:

$$\pi(a|s) = \begin{cases} 1 - \epsilon, & \text{if } a^* = \arg \max_{a \in A} Q(s, a) \\ \epsilon, & \text{otherwise take a random action} \end{cases} \quad (5.16)$$

Equation 5.16 is known as the  $\epsilon$ -greedy policy (Tokic) [11]. Using the  $\epsilon$ -greedy policy results in the agent exploring occasionally, which allows for faster convergence if  $\epsilon$  is chosen appropriately, where  $\epsilon$  is a hyper-parameter for the Model-free control algorithms. Usually, an  $\epsilon$  of 0.1 is a good starting value.

### 5.2.2. SARSA: On-policy TD Control

Model-free control can be implemented in two ways. On-policy learning, which lets the agent learn characteristics of a policy  $\pi$  by sampling from  $\pi$  and off-policy learning, learning characteristics of policy  $\pi$  by sampling another policy.

We now look at using TD methods for control. Although a Monte Carlo control method also exists, we will not use it because Monte Carlo methods have to wait until the end of an episode to update. The name of the SARSA algorithm comes from the sequence

of state-action-reward-state-action pairs. Algorithm 4 below, adapted from Sutton and Barto [1], shows the pseudo-code which can be used to implement SARSA.

---

**Algorithm 4:** SARSA algorithm to find optimal state action function  $Q_*(S, A)$

---

function SARSA ( $\alpha, \gamma, \epsilon$ );

**Input** : Three nonnegative fractions between 0 and 1, learning rate  $\alpha$ , discount  $\gamma$  and exploration probability  $\epsilon$

**Output** :  $Q(S, A)$

- 1: Initialize  $Q(s,a)$ ,  $\forall s \in S, a \in A(s)$  arbitrarily and  $Q(\text{terminal state}, \{\text{all actions}\}) = 0$
  - 2: for each episode:
  - 3:     Initialize  $S$
  - 4:     Choose  $A$  from  $S$  using  $\epsilon$ -greedy policy derived from  $Q$
  - 5:     for each step in episode:
  - 6:         Take action  $A$ , observe  $R, S'$
  - 7:         Choose  $A'$  from  $S'$  using  $\epsilon$ -greedy policy derived from  $Q$
  - 8:          $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma Q(S',A') - Q(S,A)]$
  - 9:          $S \leftarrow S'; A \leftarrow A';$
  - 10:     Until  $S$  is terminal
- 

By following the same process that we used in the TD prediction section, we can arrive at an equation similar to Equation 5.6, seen in line 8 of Algorithm 4. Instead of updating the state-value function  $V(S)$ , we update the action-value function  $Q(S, A)$ . For every state action pair, we can calculate the action-value function  $Q(S,A)$  using TD learning seen in line 8 of Algorithm 4, where  $R + \gamma Q(S', A')$  is the TD target. Lines 4 and 7 in algorithm 4 is policy improvement.

Sutton and Barto [1] state that SARSA always converges towards an optimal action-value function, as long as all state-action pairs are visited enough times. This also follows from the law of large numbers, as was stated in the Monte Carlo prediction section.

### 5.2.3. Q-learning: Off-policy TD Control

Another algorithm that we can use for model-free control is Q-learning. It is also a TD method, but it is off-policy. The optimal action-value function  $q_*(s, a)$  can be found using the Q-Learning algorithm directly without any dependency on the policy  $\pi$  being followed. We only use the policy  $\pi$  to select the next state-action pair from the current



state. Therefore, Q-Learning is an off-policy method.

---

**Algorithm 5:** Q-learning algorithm to find optimal state action function  $Q_*(S, A)$

---

function Q-learning ( $\alpha, \gamma, \epsilon$ );

**Input** : Three nonnegative fractions between 0 and 1, learning rate  $\alpha$ , discount  $\gamma$   
and exploration probability  $\epsilon$

**Output** :  $Q(S, A)$

- 1: Initialize  $Q(s,a)$ ,  $\forall s \in S, a \in A(s)$  arbitrarily and  $Q(\text{terminal state}, \{\text{all actions}\}) = 0$
  - 2: for each episode:
  - 3:     Initialize  $S$
  - 4:     for each state in episode:
  - 5:         Choose  $A$  from  $S$  using policy derived from  $Q$  (eg.,  $\epsilon$ -greedy)
  - 6:         Take action  $A$ , observe  $R, S'$
  - 7:          $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \arg \max_{a \in A} Q(S',a) - Q(S,A)]$
  - 8:          $S \leftarrow S'$ ;
  - 9:     Until  $S$  is terminal
- 

The reason the update in line 7 of Algorithm 5 is chosen as seen, is that in line 7 of Algorithm 5 the action-value function can be seen updated similar to SARSA, where we use a TD target of  $R + \gamma \arg \max_{a \in A} Q(S', a)$ . The main difference between SARSA and Q-learning is that in Q-learning we do not sample a next action but rather select one based on the current Q-function. Using Q-learning, allows both the behavior policy  $\pi$  and the target policy ( $\arg \max_{a \in A} Q(S', a)$ ) to improve. The reason we choose the TD target to update in this manner is so that the agent can learn about the optimal policy while following an exploratory policy.

In Chapter 3, MDPs were used to formulate the reinforcement learning problem. In Chapter 4, we looked at using planning to solve the problem. In Chapter 5 we looked at model-free learning, in the order of first prediction and then control. The two algorithms we will use going forward are SARSA and Q-learning.

# Chapter 6

## Applying Reinforcement Learning to Sliding Puzzles

The goal of this chapter is to solve the a 3x3 puzzle and describe how we do so. All work generated in this chapter is done using a Lenovo S145, with an i3 processor. The code was self typed in python. No external libraries were used besides the native python modules numpy, itertools and matplotlib. The code can be found at the public Github repository at the link: <https://github.com/umr-bot/sliding-puzzle-solver-bot>.

### 6.1. Representing 2x2 Puzzles

The first step in applying reinforcement learning to solve our sliding puzzle problem, requires us to have a method to encode our problem into the form of an MDP. The 2x2 sliding puzzle is small, with only  $4! = 24$  possible states, which is why we first discuss its encoding before the 3x3 puzzle which we will later do. We now describe how we encoded the 2x2 sliding puzzle into an MDP.

We represent the puzzle the same way as we introduced in Chapter 1, by reading the tile numbers row by row from left to right. Using Figure 6.1a as an example, we would read it as  $[3,1,4,2]$ . The terminal state of the 2x2 sliding puzzle problem, we define as the puzzle which reads as  $[1,2,3,4]$ , where number 4 represents the blank tile. There are two ways in which we can represent the states. The first way is in vector form, where we treat the state space as a vector containing all the states, with each state unique. This method of interpreting the states is useful for human readability.

The second way we can represent the states is by assigning each state an index from 1 to  $N^2!$ , where N is the width and height of the puzzle. We treat each state as a 4 digit number for the 2x2 puzzle and then order the states in ascending order of their respective numbers. This method of arranging the states is what we let the algorithm use.

We now define the way we encode the actions. The actions are defined as the possible directions the blank tile can move. We encode our 4 possible actions up, down left and right numerically as follows:

$$a_{up} = 0; \quad a_{down} = 1; \quad a_{left} = 2; \quad a_{right} = 3$$



**Figure 6.1:** Sequence of 2x2 sliding puzzle states solved from an arbitrary starting state seen in 6.1a. The goal of the game is to arrange the tiles in ascending order read row by row, left to right with the blank tile in the bottom right corner as seen in 6.1d. The rules are described in detail in Chapter 1.

We now have a way to encode the states and actions for the 2x2 puzzle and still need to choose what our reward function should be. We choose the reward function to be a constant of  $R=-1$  for every move the agent makes. We choose -1 so that we can calculate the amount of moves the agent took by looking at the sum of the rewards. At the start of the agents training, we initialize the state-action function  $Q(s,a)=0$ , which makes the agent see all state-action pairs as equally valuable. In order to make the agent see the state-action pairs which leads to solving the puzzles as more valuable, we reward the agent with a reward much larger than -1, when it solves a puzzle. We choose this reward as 100. The reward of 100 was experimentally determined, by having observed the maximum amount of moves the agent took to solve the puzzle (about 40 moves), which we will later see plotted. Our reward function can now be symbolically written as:

$$R = \begin{cases} -1, & \text{every time agent takes an action} \\ 100, & \text{if agent reaches terminal state} \end{cases} \quad (6.1)$$

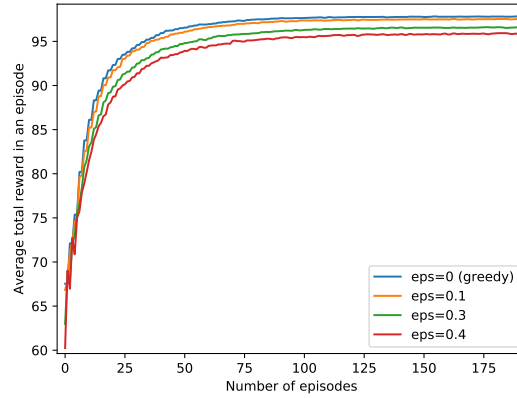
### 6.1.1. Hyper-parameter Selection for 2x2 Sliding Puzzle

In this section we will look at the learning curves of the agent trained to solve the 2x2 puzzle, for varying values of the hyper-parameters. When we train an agent, we let it solve many puzzles, from arbitrary start states. All states that come between the initial and terminal state is defined as an episode. We choose the amount of episodes to train the agent with by plotting the learning curve. In order to obtain a smooth learning curve, we train the agent multiple times for the same amount of episodes and plot the average of the total reward that the agent obtains in each episode. Each training set we call a run.

Figure 6.2 shows the learning curves of an agent trained to solve a 2x2 sliding puzzle using SARSA implemented from Algorithm 4. We plot the curves for different values of  $\epsilon$  so that we can compare which value of  $\epsilon$  works best. Recall that the  $\epsilon$ -greedy policy was defined as:

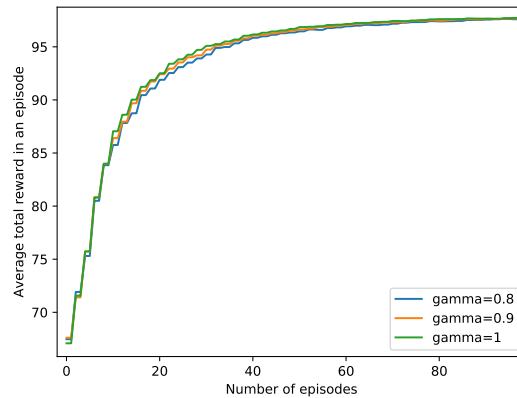
$$\pi(a|s) = \begin{cases} 1 - \epsilon, & \text{if } a^* = \arg \max_{a \in A} Q(s, a) \\ \epsilon, & \text{otherwise take a random action} \end{cases}$$

We can see in Figure 6.2 that using  $\epsilon = 0$  yields the best training of the 2x2 sliding puzzle agent. When  $\epsilon = 0$  the agent follows the greedy policy where the agent only takes the



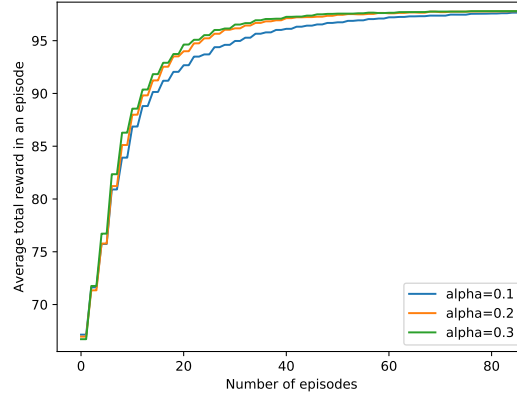
**Figure 6.2:** Learning curve for agent training using SARSA for different values of  $\epsilon$ , we keep constant  $\gamma = 0.9$  and  $\alpha = 0.1$ . The plot shows the total reward in an episode, averaged over 10000 samples. By sample, we refer to running the algorithm once and retrieving the total reward in each episode. We choose  $\alpha$  and  $\gamma$  arbitrarily.

actions which maximizes  $Q(s,a)$ , which can also be seen if we substitute  $\epsilon$  into the  $\epsilon$ -greedy policy equation. The reason that the greedy policy works best for the 2x2 sliding puzzle case, is that the reward function  $R$ , is deterministic and the state space is very small. What we can also note is that the total reward obtained by the agent in an episode converges by 100 episodes. We now also plot the learning curves for different values of  $\alpha$  and  $\gamma$ .



**Figure 6.3:** 2x2 puzzle SARSA learning curve.  $\gamma$  is varied while  $\alpha = 0.1$  and  $\epsilon = 0$

Figure 6.3 shows the learning curve for different values of the discount  $\gamma$  when using SARSA to solve the 2x2 puzzle. We only plot a few values to make the graph clearer to read. We also use  $\epsilon = 0$  as we earlier saw that it provides the fastest convergence of the learning curve. We can see that  $\gamma = 1$  yields the fastest convergence, but the difference between  $\gamma = 1, \gamma = 0.9$  and  $\gamma = 0.8$  is not much. We can reason that the reason that  $\gamma = 1$  performs best is that the 2x2 puzzle is small and also deterministic. We also then plot the learning curve for different values of the learning rate  $\alpha$ . Recall that the update function



**Figure 6.4:** 2x2 puzzle SARSA learning curve.  $\alpha$  is varied while  $\gamma = 1$  and  $\epsilon = 0$

in the SARSA algorithm is:

$$Q(S, A) = Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)].$$

The value of  $\alpha$  determines how much we update  $Q(s,a)$  towards the latest information that we receive from the state-action pair  $(S', A')$ . When  $\alpha$  is larger we get a faster converging learning curve, but if it is too large the curve will never converge. We only plot a few values of  $\alpha$  in Figure 6.4 to point out which values we found were optimal for solving the 2x2 puzzle.

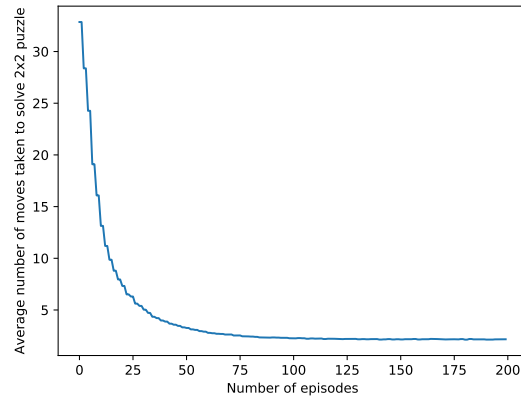
We now describe how we calculated the amount of moves the agent took on average to solve the puzzle. The total reward in an episode can be given by:

$$R_{total} = \sum_{k=0}^{\infty} R_{k+1} \quad (6.2)$$

We know that the agent obtains a reward of -1 for each action and 100 when it solves a puzzle, hence we can calculate the number of actions the agent took in a episode as:

$$\#actions = 100 - R_{total}. \quad (6.3)$$

We now use Equation 6.3 to plot the number of actions that the agent took to solve the 2x2 sliding puzzle while the agent was being trained. Figure 6.5 shows the average number of actions the agent took while training to solve the 2x2 puzzle. We used SARSA with the optimal hyper-parameter values we earlier found, which are  $\epsilon = 0$ ,  $\gamma = 1$  and  $\alpha = 0.1$ . We can see that the agent learns the optimal Q-value function  $Q(s,a)$  at after about 100 episodes, which confers with what we saw in the hyper-parameter varying figures. The trained agent approximately takes 2 to 3 moves to solve the 2x2 sliding puzzle. This number of moves makes sense, because if the agent acts optimally, then the maximum amount of moves that it should take to solve a 2x2 puzzle is 6 moves. In the next section



**Figure 6.5:** This plot shows the number of moves the agent takes to solve the 2x2 puzzle each episode. As the agent trains, we can see that it takes less moves to solve the puzzle. The agent was trained using SARSA with  $\epsilon = 0$ ,  $\gamma = 1$  and  $\alpha = 0.1$ .

we explain how we solved the 3x3 puzzle, where we use a similar encoding to that which we used for the 2x2 puzzle.

## 6.2. Representing 3x3 Puzzles

We now use a similar encoding on the 3x3 puzzle, with the only difference being the state space. We now use the same encoding method we used for the 2x2 puzzle on the 3x3 puzzle. In the 2x2 puzzle we represented each state by an index 1 to  $N^2!$ , where  $N$  was the horizontal and vertical number of tiles of the puzzle. For the 3x3 puzzle there are  $9!$  states and hence we need a table with  $9!$  indices to store the states. We again treat the states as numbers where we read the tiles of the puzzle row by row from left to right. Now we will represent the blank tile by the number 9. The states if treated as a number, are ordered in ascending order in the Q-table.

For clarity we show some of the states in the 3x3 puzzles Q-table as follows:

$$\begin{aligned}
 [1,2,3,4,5,6,7,8,9] &\rightarrow \text{state 1} \\
 [1,2,3,4,5,6,7,9,8] &\rightarrow \text{state 2} \\
 [1,2,3,4,5,6,9,7,8] &\rightarrow \text{state 3} \\
 &\cdot \\
 &\cdot \\
 &\cdot \\
 [9,8,7,6,5,4,3,2,1] &\rightarrow \text{state 362880}
 \end{aligned}$$

### 6.3. Sequential Approach for 3x3 Puzzles

Only using the tabular method to solve the 3x3 puzzle will take too long, we therefore devise a method that allows us to reduce the amount of training time by looking at how humans solve the puzzle. The method which we found always results in the puzzle being solved. We now detail what this method is and how we use reinforcement learning to utilize it to make the training of the agent much faster. The human inspired method consists of solving the puzzle by sliding in place the first row and first column first, then the second row and column, then the third row and third column. In other words we first slide tile number 1, 2 and 3 in place. Next we slide tiles number 4 and 7 in place without moving any of the top row tiles. Then the puzzle is finished being solved when we solve the 2x2 grid in the right bottom corner.

Five agents are used to solve the different parts of the puzzle. Each agent will only solve one part of the puzzle. The reason we do so is that the problem after being broken up is much more tractable and easier to solve. Agent 1 trains to place tile number 1 in the top left corner of the puzzle grid. Figure 6.6 shows the way Agent 1 sees the puzzle. The tiles with an X in can have numbers from 2-8. There are only three unique tiles in Agent 1's view of the puzzle, with nine possible positions that each tile can be in. Hence the amount of possible states for Agent 1 is  $9*8*7 = 504$  states.

1	X	X
	X	X
X	X	X

**Figure 6.6:** Puzzle size that Agent 1 sees and trains to solve. The tiles with X's can have any value from 2-8

Agent 2 only looks at the last two rows of the 3x3 puzzle and does not move any tiles in the top row. The terminal state that Agent 2 solves towards is shown in Figure 6.7. The purpose of Agent 2 is to move the blank, number 2 and number 3 tiles into the last two columns of the 3x3 puzzle. This is done as Agent 3 only looks at the last two columns to move the number 2 and 3 tiles into the first row. There are three unique tiles in Figure 6.7 with six possible positions. Therefore the amount of states that Agent 2's Q-table has is  $6*5*4 = 120$  states.

Figure 6.8 shows the puzzle configuration that Agent 3 trains on. If Agent 2 had not moved the blank, number 2 and number 3 tiles into the last two columns, then Agent 3 would not have all the tiles it requires to solve to its terminal state. The terminal state of Agent 3 is reached when the blank, number 2 and number 3 tiles are in the positions shown in Figure 6.8. Agent 3 has the same number of states as Agent 2 which is 120 states.

X		2
X	3	X

**Figure 6.7:** Puzzle size that Agent 2 sees and trains to solve. The puzzle in this figure represent the last two rows of the 3x3 puzzle.

Note similarly to Agent 2, Agent 3 does not move any tiles in the first column, as it does not see that part of the 3x3 puzzle. The same logic will also apply to Agents 4 and 5.

2	3
	X
X	X

**Figure 6.8:** Puzzle size that Agent 3 sees and trains to solve. The puzzle in this figure represent the last two columns of the 3x3 puzzle.

Agent 4's terminal state can be seen in Figure 6.9. Agent 4 has the same amount of states as Agent 2 and Agent 3 which is 120 states.

4	X	X
7		X

**Figure 6.9:** Puzzle size that Agent 4 sees and trains to solve. The puzzle in this figure represent the last two rows of the 3x3 puzzle.

Figure 6.10 shows the puzzle that Agent 5 must solve. Agent 5 solves the 2x2 grid of tiles that is left after having used Agent 1, 2,3 and 4 to slide the first row and column into their final positions. The 2x2 puzzle is discussed in detail at the beginning of this chapter, where it was already stated that it has 24 states.

5	6
8	

**Figure 6.10:** Puzzle that Agent 5 sees and trains on. The puzzle in this figure represents the bottom right grid of tiles in the 3x3 puzzle.



If all five agents numbered 1-5 are used in sequence, then the sliding puzzle is always solved if it is a solvable puzzle. An algorithm of how the 5 Agents can be used in sequence is shown in Algorithm 6.

---

**Algorithm 6:** Pseudo-code showing how five agents can be used to solve a 3x3 puzzle

---

**Input** : A sliding puzzle in the form of an array named puzzle

**Output**: Boolean True or False, if puzzle is solved

```

1: if puzzle[0] not equal to tile 1:
2:     Call Agent 1
3: if puzzle[3] or puzzle[6] equals either blank tile or tile 2 or tile 3:
4:     Call Agent 2
5: if puzzle[2] not equals tile 2 and puzzle[3] not equals tile 3:
6:     Call Agent 3
7: if puzzle[3] not equal to 4 or puzzle[6] not equal to 7:
8:     Call Agent 4
9: if puzzle[4] not equal tile 5 or puzzle[5] not equal tile 6
10:    or puzzle[7] not equal tile 8 or puzzle[8] not equal blank tile:
11:    Call Agent 5
12:    if Agent 5 reaches terminal state:
13:        return True
14:    else return False

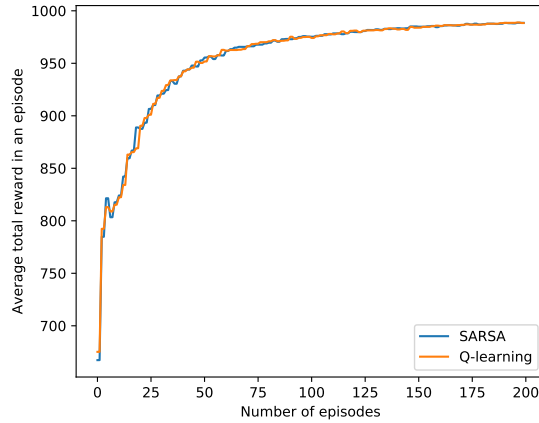
```

---

For the value of a given state to be accurate, the agent must encounter that state many times. If the state space is larger, the probability that a certain sequence of states will be seen becomes much lower. Hence it is better to have a smaller state space, in order for the agent to learn the optimal policy faster. If we use all agents (1 to 5 in ascending order), then the total amount of states that we have are  $504 + (3 \cdot 120) + 24 = 888$  states. This is much smaller than if we had used tabular reinforcement learning, which would require us to deal with  $9! = 362880$  states for the 3x3 sliding puzzle. We will now look at results of the individual agents being trained for different hyper parameter values.

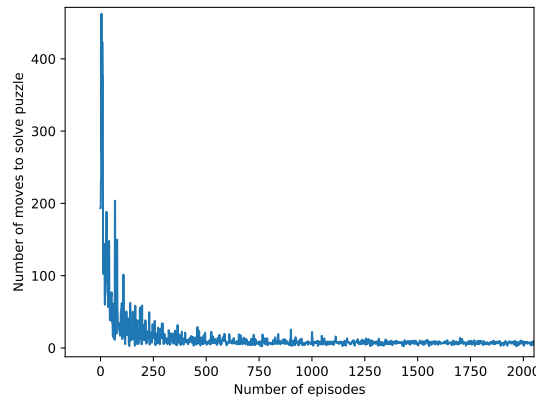
## 6.4. Experiments and Results

In the following section the SARSA and Q-learning algorithms are used to train the agents mentioned in the previous section. Later we will compare SARSA and Q-learning to see which one is better with regards to the amount of moves the agent takes to solve the puzzle after being trained. The agents of the 3x3 puzzle have the same state-space size as the 2x2 puzzle. Hence we will from here on use the hyper-parameters that we saw works best for the 2x2 puzzle.



**Figure 6.11:** SARSA vs Q-learning for Agent 1.

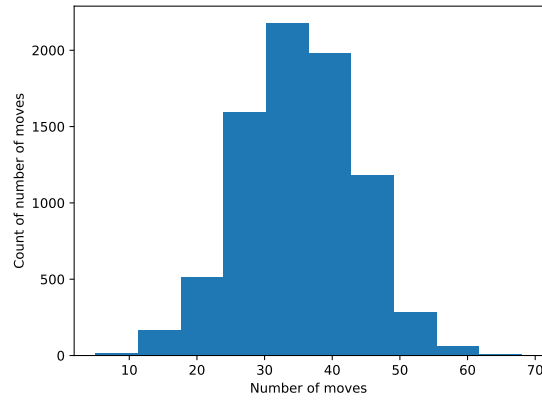
In Figure 6.11 we show the reward learning curve for SARSA vs Q-learning, averaged over a 1000 runs. We can see that SARSA and Q-learning results in almost the same curve. This is most probably due to the reason that we have a deterministic environment, which is defined through the reward function  $R$ . We will use SARSA from here on to detail the results of the combined Agents 1 to 5.



**Figure 6.12:** The amount of moves that the combined 3x3 agent takes to solve the puzzle. The results are taken from the beginning to the end of the SARSA training.

We can see that the 3x3 agents training converges by about 300 episodes. This makes sense as we saw for the 2x2 agent, which has a similar sized state space to the agent of the 3x3 puzzle, that it took about 100 moves for its policy to converge to the optimal policy.

Figure 6.13 is generated using data from solving 8000 puzzles. Figure 6.13 depicts an histogram which shows the counts of how many moves the agent took to solve the puzzle. What we can see is that on average, the agent takes about 35 moves to solve the puzzle, with a minimum and maximum amount of moves of 10 and 60 moves respectively. The 8000 puzzles were solved in less than 10 seconds. According to Culberson [7], the 3x3 sliding puzzle can always be optimally solved in less than 80 moves for all configurations.



**Figure 6.13:** The histogram which shows the counts of the amount of moves that the combined 3x3 agent takes to solve the puzzle, while the agent is trained using SARSA.

If we consider that the maximum amount of moves our puzzle takes is close to 60 moves and an average of 35 moves, then we can say that our solution is close to optimal. In order to actually obtain an optimal solution, we would have to use another method to solve the puzzle such as the search algorithms used by Korf [6] and Culberson [7]. Although we cannot compare directly to Korf's [6] solution where they solved the 5x5 puzzle in the time between two and half hours to over a month, depending on the puzzle configuration. It seems that our solution is faster than theirs, as we only use a few seconds to find a solution on average using python code.

# Chapter 7

## Summary and Conclusion

In this report we solved a 3x3 sliding puzzle using reinforcement learning. We first looked at alternate methods to solving sliding puzzles in Chapter 2. We then described in Chapter 3, the theory of Markov decision processes. Next in Chapter 4 we looked at using dynamic programming to solve Markov decision processes. In Chapter 5 we then built on the dynamic programming theory and used it to define reinforcement learning techniques. In Chapter 6 we used the Markov decision process, dynamic programming and reinforcement learning theory to solve a 3x3 puzzle. We detailed how we did so by first describing how we solved a 2x2 puzzle and then expanded the methodology to solve the 3x3 puzzle. We then inspected which reinforcement learning algorithms worked better, as well as how many moves the algorithms took to solve the puzzles.

The aim of this project was to solve a sliding puzzle with the minimum amount of moves. We have achieved this aim and now have a well defined method to solve a 3x3 sliding puzzle with an optimal amount of moves most of the time. The number of moves are not always optimal as we did not use purely tabular (brute force) reinforcement learning, but instead partially guided the way the training was executed. On average the agent we trained solves the 3x3 puzzle in 35 moves and maximally in plus minus 60 moves. Using our designed method, the agent solves a given puzzle with a 100% success rate, if the given puzzle is solvable. Using SARSA and Q-learning to train our agent took less than 10 seconds, which is relatively short and meets the project objective of speed.

As future work one can further expand the methodology that we used to solve the 3x3 puzzle to solve a 4x4 puzzle. This can be done by treating a 4x4 puzzle as a column and a row plus a 3x3 puzzle. Similar to how we solved the 3x3 puzzle, where we treated it as a 2x2 puzzle plus a row and a column. Other avenues of research include using function approximation to train an agent to solve the 3x3 puzzle. Function approximation tries to find an underlying function to observations that are obtained from the environment. This can be done by using a linear function or neural network and can be used as a method for reducing the dimension of the state space (Sutton and Barto) [1].

# Bibliography

- [1] R. S. Sutton and A. G. Barto, *Reinforcement Learning An Introduction second edition*. The MIT Press, 2018.
- [2] P. U. F. of Computer Science, “8puzzle assignment.” [Online]. Available: <https://www.cs.princeton.edu/courses/archive/spring18/cos226/assignments/8puzzle/index.html>
- [3] A. F. Archer, “A modern treatment of the 15 puzzle.” pp. 793–799, (1999).
- [4] A. Géron, “Hands-on machine learning with scikit-learn and tensorflow.”
- [5] E. Hordern, “Sliding piece puzzles.” 1986.
- [6] L. A. T. Richard E. Korf, “Finding optimal solutions to the twenty-four puzzle,” 1996.
- [7] J. S. Joseph C. Culberson, “Efficiently searching the 15- puzzle. technical report 94-08 (unpublished),” 1994.
- [8] R. Spaans, “Solving sliding-block puzzles,” 2009.
- [9] P. Weng, “Markov decision processes with ordinal rewards: Reference point-based preferences,” 2011.
- [10] D. Silver, “Ucl course on rl,” 2015. [Online]. Available: <https://www.davidsilver.uk/teaching/>
- [11] M. Tokic, “Adaptive  $\epsilon$ -greedy exploration in reinforcement learning based on value differences,” 2014.
- [12] W. E. Johnson, Wm. Woolsey; Story, “Notes on the ”15” puzzle,” *American Journal of Mathematics*, vol. 29, no. 2(4), p. 397–404, 1879.
- [13] D. of the Cube Forum, “5x5 sliding puzzle can be solved in 205 moves,” 2016. [Online]. Available: <http://cubezzz.dyndns.org/drupal/?q=node/view/559>
- [14] Minsky, *Computation: finite and infinite machines*. Prentice Hall, 1967.

# Appendix A

## Project Planning Schedule

Week	Date	Task
1	05/15/20	Identify problem
2	05/22/20	Read up on RL theory
3	05/29/20	Look at David Silvers lecture series
4	06/05/20	Look at David Silvers lecture series
5	06/12/20	Implement grid-world example from David Silvers series
6	06/19/20	Implement grid-world example from David Silvers series
7	06/26/20	Implement grid-world example from David Silvers series
8	07/03/20	Start covering model-free prediction as a prelude model-free control
9	07/10/20	Start with model-free control: SARSA and Q-learning
10	07/17/20	Start applying theory in proper to solve sliding puzzle
11	07/24/20	Continue writing environment for 2x2 puzzle RL agent
12	07/31/20	Complete 2x2 puzzle RL agent
13	08/07/20	Expand 2x2 puzzles RL environment to fit 3x3 puzzle
14	08/14/20	Complete 3x3 puzzle RL agent
15	08/21/20	Start writing report
16	08/28/20	Continue with report writing and fix any bugs if left in code
17	09/04/20	Finish writing report
18	09/11/20	Use feedback from lecturer to refine report

# Appendix B

## Outcomes Compliance

ELO assessment	How the ELO was met
ELO 1: Problem solving	The problem was identified in the introduction, where the objectives were also formulated. The problem was solved and analyzed in Chapter 6.
ELO 2: Application of scientific and engineering knowledge	Knowledge of mathematics was used to describe the theory necessary to understand RL. Engineering knowledge was used to apply RL to the problem.
ELO 3: Engineering Design	The solution to the problem was designed using a creative method. The method was applied in a procedural manner.
ELO 4: Investigations, experiments and data analysis	In Chapter 6 investigations and experiments were conducted.

Figure B.1

ELO 5: Engineering methods, skills and tools, including Information Technology	Chapter 6 details all experiments done, which required the use of software development skills.
ELO 6: Professional and technical communication:	This ELO is met by the written report and the presentation that is to be done.
ELO 8: Individual work	All code that was used in chapter 6 was written by the author (excluding some native libraries mentioned in Chapter 6) with the help of the supervisor. The report was self written.
ELO 9: Independent Learning Ability	Reinforcement learning was not covered at all in the undergraduate degree and is also a field that is still under ongoing research. The author had to learn about a new field of study in order to carry out this project.

**Figure B.2**





# Appendix C

## Student and Supervisor agreement

Agreement between skripsie student and study leader regarding mutual responsibilities

*Project (E) 448, Department of Electrical and Electronic Engineering, Stellenbosch University*

Student name and SU#:	Umr Barends 18199313
Study leader:	Mr JC Schoeman
Project title:	Learning to solve Sliding Puzzles using Reinforcement Learning
Project aims:	Generally in robotics a manipulator (eg. an arm) is used to manipulate an object in the environment. It is normally easier to first simulate the robots behavior in a more simple environment. In this project we try to solve a sliding puzzle using reinforcement learning, where the same algorithm can then later be applied to the robotics problem.
<ol style="list-style-type: none"><li>1. It is the responsibility of the student to clarify aspects such as the definition and scope of the project, the place of study, research methodology, reporting opportunities and -methods (e.g. progress reports, internal presentations and conferences) with the study leader.</li><li>2. It is the responsibility of the study leader to give regular guidance and feedback with regard to the literature, methodology and progress.</li><li>3. The rules regarding handing in and evaluation of the project is outlined in the Study Guide/website and will be strictly adhered to.</li><li>4. The project leader conveyed the departmental view on plagiarism to the student, and the student acknowledges the seriousness of such an offence.</li></ol>	
Signature — study leader:	
Signature — student:	
Date:	5 August 2020

(Upload this form to SUNLearn)

**Figure C.1:** Student and Supervisor agreement