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

October 29, 2020

Acknowledgements

I would like to thank my dog, Muffin. I also would like to thank the inventor of the incubator; without him/her, I would not be here. Finally, I would like to thank Dr Herman Kamper for this amazing report template.

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 woordelikse 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 aange-dui, 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 / *October 29, 2020*

Abstract

English

The English abstract.

Contents

List of Figures

List of Tables

Nomenclature

Variables and functions

$p(x)$ Probability density function with respect to variable x.

Acronyms and abbreviations

RL   Reinforcement Learning

SARSA   State-Action-Reward-State=Action

Q-learning   Q value learning

MDP   Markov Decision Process

Chapter 1

Introduction

1.1. Problem Statement and Project Objective

In robotics, a manipulator (ie. 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 doing developing reinforcement learning algorithms for robotics is to first solve similar, less complex problems.

In this project reinforcement learning is used to solve sliding puzzle. An example of a sliding puzzle being solved can be seen in Figure 1.1. The objective of this project is to solve a shuffled puzzle with the minimum amount of moves using reinforcement learning. The rules of the game are explained in the next section.

1.2. Background

The sliding puzzle is a game with a history dating back to the 1870's [3]. The game consists of $N^2 - 1$ tiles arranged on an NxN grid with one empty tile. The tiles are numbered 1 to N. A possible configuration of a 3x3 puzzle is shown in Figure 1.1a. The puzzles 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. This can visually be seen between Figures 1.1a and 1.1b where tile number 6 moves right in the transition between Figures 1.1a and 1.1b. We will denote the action that can occur in a certain state by the possible movements the empty tile can move. Which will be defined as $A_s = up, down, left, right$.

The sliding puzzle is a game with a history dating back to the 1870's [3]. Many solutions have been thought up to solve such puzzles, which include using search algorithms [4]. The rules of the game are that the only tiles that can move are the ones next to the tile with no number, known as the 'blank tile'. Additionally only one tile can be moved at a time. The final puzzle configuration is shown in 1.1d, 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.

(a) Puzzle with tiles 5,6,8 and blank in the incorrect positions

(b) Puzzle with tiles 5,8 and blank in the incorrect positions

(c) Puzzle with tiles 8 and blank in the incorrect positions

(d) Solved puzzle

Figure 1.1: Figures of a 3x3 sliding puzzle being solved

1.3. Document Outline

In Chapter 2 we will look at a paper which used an alternate approach to solve a 16 tile sliding puzzle. Chapter 3 describes in detail theory relating to Markov decision processes. Chapter 5 discusses in detail the dynamic programming and reinforcement learning techniques which will be used to solve our puzzle problem.

In Chapter 6 the theory of reinforcement learning is applied to solve sliding puzzles. In the same chapter we

Rather how you used the theory of RL to design the solution to the sliding puzzle problem. Do this mathematically. Talk about things like the size of the state space, the sequential agents that you trained, how you decided on algorithms or hyperparameters.

Chapter 7 contains tests and verification that our solution works using the results of various experiments. Chapter 8 concludes the report.

Chapter 2

Literature Review

2.1. Introduction

Reinforcement learning is a method of learning what to do by linking states to actions with a numerical reward incurred for every state-action pair. The final goal of RL is to find a path of states and actions with the maximum amount of reward. [5] This path is typically described as a policy π.

learning and planning lecture slide 42 (37/46)

2.2. Solving sliding puzzle using value iteration

In this section a few terms will be used which are fully described in the theory chapters 3 and 5. These terms include value iteration and policy iteration.

A 4x4 puzzle solution is investigated in [6]. The method they use is a variation of value iteration. The amount of states in a 4x4 puzzle is approximately $10\hat{1}3$ elements. It was found in [6] that value iteration is indeed a feasible approach if the value iteration algorithm is altered so that the only a subset of the entire state space is used sequentially until a solution is found.

This reducing of the state space is similar to what will later be done in this report. However the method we use to reduce the state space is by having our algorithm be guided somewhat by how a human would solve a puzzle, by doing the edges first. Additionally that we use a different reinforcement learning method known as policy iteration.

Solutions to sliding puzzles can also been found by using search algorithms such as A* and IDA* [4].

Chapter 3

Markov Decision Processes

3.1. Introduction

In order to describe reinforcement learning techniques, we first have to have a way of mathematically defining reinforcement learning problems. In this chapter, we do so by formally introduce the concept of Markov decision processes (MDPs). This chapter introduces the key concepts required to address the reinforcement learning problem. These concepts includes, 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 [5].

3.2. Definition of a MDP

A Markov decision process is a stochastic, discrete, time controlled process. What this means is that a MDP is a partially controlled and partially random process [5]. In general RL problems can be described using MDPs [1]. Paul Weng 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 γ [7]. These variables can be denoted in a 5-tuple as

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

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 an MDP, the assumption is that all the states have the Markov property [5]. This means that

$$P[S_{t+1}|S_t] = P[S_{t+1}|S_1, ..., S_t].\tag{3.2}$$

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.

3.3. 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 γ. The discount $\gamma \in [0,1]$ determines the present value of future reward R_t.

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... \\ &= \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \end{aligned}\tag{3.3}$$

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 $\gamma = 1$ [5]. What can also note is that using a discount factor between 0 and 1 makes the return finite because R_{t+k+1} is finite and

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

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

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

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

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

$$v(s) = E[G_t|S_t = s].\tag{3.6}$$

This equation shows that the value function v(s) is the expected return given that v(s) is evaluated in state s at time-step t.

3.4. Bellman Expectation Equation

Sutton and Barto [5] decomposes v(s) so that it becomes a recursive function as follows

$$v(s) = E[G_t|S_t = s] \tag{3.7}$$
$$= E[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...|S_t = s] \tag{3.8}$$
$$= E[R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + ...)|S_t = s] \tag{3.9}$$
$$= E[R_{t+1} + \gamma G_{t+1}|S_t = s] \tag{3.10}$$
$$= E[R_{t+1} + \gamma v(S_{t+1})|S_t = s]. \tag{3.11}$$

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. [5] Sutton and Barto then defines the value function given that the agent follows the policy π as

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

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

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

The q-function is defined as the expected return of taking action a from state s and thereafter following policy π. 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 [5] decomposes 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

$$v_\pi(s) = E_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s] \tag{3.14}$$
$$= \sum_{a' \in A} \pi(a|s)(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_\pi(s')) \tag{3.15}$$

and

$$q_\pi(s,a) = E_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1})|S_t = s, A_t = a] \tag{3.16}$$
$$= R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \sum_{a' \in A} \pi(a'|s')q_\pi(s',a'). \tag{3.17}$$

Equation 3.15 can be placed into vector form as:

$$v_\pi = R^\pi + \gamma P^\pi v_\pi \tag{3.18}$$

Which has the solution:

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

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

3.5. Optimal Policies and Bellman Optimality Equations

Sutton and Barto now defines the optimal policy, which only has value 1 if the action the agent takes maximizes $q_*(s, a)$ [5]. This is mathematically defined as:

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

There always exists an optimal deterministic policy for any MDP [5]. 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. Sutton and Barto [5] then defines what is know as the Bellman *optimality* equations.

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

and

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

What equation 3.21 represents is the maximum return that can be obtained from state s, when following the optimal policy π_*. Equation 3.22 represents the maximum return that can be obtained by taking action a state s and thereafter following the policy. 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

4.1. Background

Dynamic programming (DP) is a method to solve problems by using a divide and conquer approach [1]. That is to say a problem is broken into smaller sub-problems and then solved. Dynamic programming is used to solve MPDs 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, theoretically DP is still important and necessary in reinforcement learning [5]. There are two properties that must be met for a problem to be solvable via DP: optimal substructure and overlapping sub-problems. [1]

- 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 satisfies both of these conditions. 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. [1] The key concept of DP and of reinforcement learning, is to use state-value functions as the structure through which optimal policies are found [5]. As a basis to finding the optimal policy, the next section hence discusses how to first evaluate a given policy.

4.2. Policy Evaluation

In order to evaluate a given policy, we calculate the state-value function V_π for a given policy π. This is called policy evaluation [5]. From equation 3.15 we know that

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

In order to make it clear how policy evaluation works, we use a diagram inspired by David Silvers lecture series [1] seen in Figure 4.1a. Starting at the root node, state s in the

diagram in Figure 4.1a, the agent can take any action based on its policy π. After the agent takes this action, the environment responds and moves the agent to state s'. The environments response depends on the probabilities contained in the function $P_{ss'}^a$.

Sutton and Barto then introduces the notation for an approximate value function, $v_k(s)$ which maps every state s $\in$ S to a real number in $\Re$ [5]. They then choose v_0 arbitrarily, where after $v_k(s)$ is successively updated using

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

The value function is updated continuously in this way, until the difference between v_k and v_{k+1} is determined negligible. This process is known as iterative policy evaluation. Now that we have a tool to evaluate a policy , in the next section we discuss how to improve a given policy, in order to find the optimal policy π_*.

(a) Policy evaluation [1]

(b) Greedy policy iteration visualisation [1]
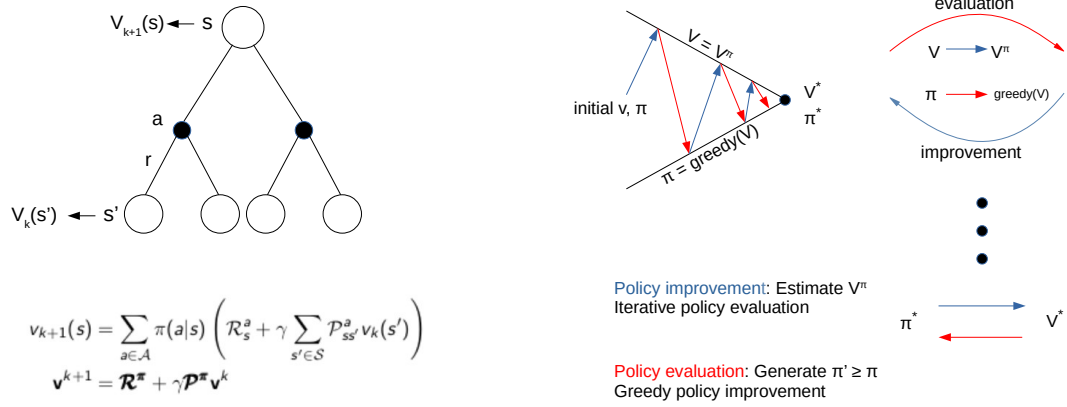
Figure 4.1: Policy iteration with images inspired by David Silvers UCL Course on RL [1]

4.3. Policy Improvement

According to David Silver, a given policy π can be improved using the following two steps [1]:

Step one is *evaluating* the policy π using equation 3.15:

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

Step two is *improving* the policy by acting greedy with respect to v_π to obtain a new policy π'.

Sutton and Barto defines acting greedy [5], as the greedy policy which makes the agent always takes the action that looks best according to v_π looking one step ahead,

$$\pi' = greedy(v_\pi). \tag{4.2}$$

Sutton and Barto then further expand the greedy policy concept to consider all states and actions resulting in

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

where $argmax_a$ denotes the action which if taken maximizes the expression following it. By using the greedy policy, Sutton and Barto proves in [5] that we will always obtain the optimal expected return v_* defined in equation 3.21.

Sutton and Barto provides a proof that using this method always results in a policy π' better, or as good as π, which will not be provided here but can be found in [5]. That is to say the expected return following π' will be greater or equal than following π

$$v_{\pi'} \geq v_\pi. \tag{4.4}$$

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

4.4. Policy iteration

There are two separate uses for dynamic programming in reinforcement learning. These are prediction and control which both have (S, A, P, R, γ) as input.

- The prediction method uses (S, A, P, R, γ) and a policy π as input, while it outputs a value function v_π. This is known as value iteration.

- The control method only takes (S, A, P, R, γ) as input and outputs the optimal value function v_* and optimal policy π_*. This is known as policy iteration.

Figure 4.1b illustrates how greedy policy iteration works. The blue lines represents policy improvement and the red lines policy evaluation. By continuously using these two steps in succession, we eventually converge to the optimal state-value function v_* and optimal policy π_* [5]. It is known that policy iteration converges, as it uses policy improvement which we know converges from the previous section [5].

4.5. Value Iteration

Policy iteration has a drawback of needing for every iteration, to do policy evaluation which in itself is a iterative computation requiring many calculations across the entire state-space [5]. Iterative policy evaluation only converges in the limit of its time steps. However Sutton and Barto questions if it is necessary to converge precisely on the optimal policy, or if we can stop reasonably short and acquire an approximate optimal policy [5]. They answer this by introducing the concept of value iteration, where they use policy iteration with a special case of policy evaluation as follows:

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

and in vector form,

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

Note these equations are the Bellman optimality equation (in equation 3.21), turned into an update rule. In the next chapter we use almost the same method to find the optimal policy, except that we do not assume that we have access to a perfect model of the MDP.

4.6. Value iteration grid-world example problem

Figure 4.2 is an example problem, which David Silver [1] uses to illustrate how value iteration works. The states are defined as cells contained within a grid. The example in Figure 4.2 has a state space of 16 states with S $\in$ [0,15] and an action space of actions A = {up, down, left, right, NONE} indicated on the figure. The action NONE is used to indicate the agent choosing to stay in its current state. Each state consists of an index and a value, with the terminal state being at index 0. The agent can move in all directions in all states, except for the terminal state, where it can only choose to stay still. The state indices are numbered from left to right and top to bottom in ascending order starting at 0. Hence we can treat the states as an array in computer programming with 0 to 3 being the top row indices, 3 to 7 the second most top row indices etc.

Imagine that we have an agent dropped into a grid as is shown in the initial iteration in Figure 4.2 then

$$v_0(0) = v_0(1) = v_0(2) = v_0(3) = v_0(4) = ... = v_0(s) = 0. \tag{4.7}$$

Hence at iteration 0 no matter which state the agent is dropped into, all actions are equally valuable to it. As the iterations k increases, $v_k(s) \rightarrow v_*(s)$. We know the agent has learnt

k = 0

O	O	O	O
O	O	O	O
O	O	O	O
O	O	O	O

k = 1

O	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1

k = 2

O	-1	-2	-2
-1	-2	-2	-2
-2	-2	-2	-2
-2	-2	-2	-2

k = 3

O	-1	-2	-3
-1	-2	-3	-3
-2	-3	-3	-3
-3	-3	-3	-3

k = 4

O	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-4
-3	-4	-4	-4

k = 5

O	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-5
-3	-4	-5	-5

k = 6

O	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-5
-3	-4	-5	-6

k = 7

O	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-5
-3	-4	-5	-6

up

left right

down
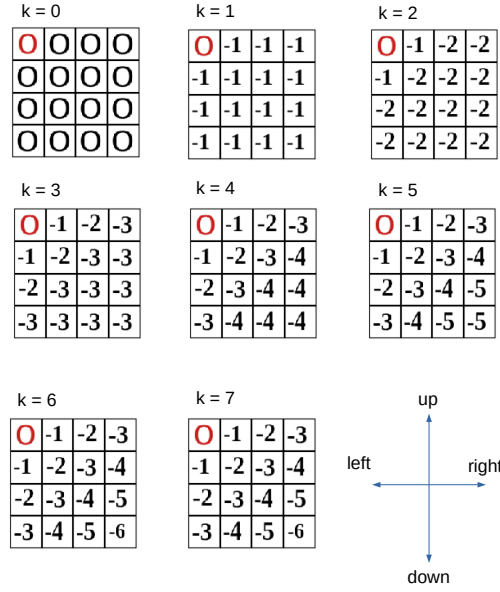
Figure 4.2: This is a grid world example solved using value iteration. The goal is to have values in the cells represent the amount of steps it would take from that cell to the cell in the top left corner of that iteration. Each step results in a reward of -1. All iterations after iteration k = 7 are the same, although not shown, as the optimal state-value function $v_7(s) = v_*(s)$ has been reached by then.

the optimal state-value function, when the current iterations state-value function has a negligible, or no difference from its previous iteration

$$v_{k+1}(s) - v_k(s) \leq \delta, \tag{4.8}$$

where δ is a chosen threshold.

Chapter 5

Reinforcement Learning

5.1. Introduction

This chapter can be divided into two sections. Model-free prediction which **estimates** an ***unknown*** MDP and model-free control which **optimizes** the value function of an ***unknown*** MDP.

5.2. Reinforcement Learning

What we have been discussing up to now all has had to do with using methods which required knowledge about the model of the environment. This knowledge was embedded in the probability transition matrix P. Often times this matrix P is extremely large making calculation times long. While at other times it is not possible to know the model of the environment beforehand. We therefore look at methods which do not need models.

There exists both model-free prediction and model-free control Reinforcement Learning techniques. Model-free **prediction** ***evaluates*** the value function of an unknown MDP. While Model-free **control** ***optimizes*** the value function of an unknown MDP.

There are two ways that model-free control can be implemented:

On-policy learning: Learning characteristics of a policy π by sampling from π

Off-policy learning: Learn characteristics of policy π by sampling another policy μ

5.2.1. Model-free prediction

Monte-Carlo Policy evaluation

Monte-Carlo policy evaluation works as follows: When time step t and state s is reached we increment a running 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 reached or incrementing N(s) *every* time states s is reached.

Then we increment the total reward to S(s) = S(s) + G_t.

Finally we calculate the value function $V(s) = \frac{S(s)}{N(s)}$

Then V(s) $\rightarrow V_\pi(s)$ as N(s) $\rightarrow \infty$

In order to incrementally do Monte-Carlo updates, an incremental mean u_k is used. The derivation for u_k from a standard mean is as follows:

$$u_k = \frac{1}{k} \sum_{j=1}^{k} x_j \tag{5.1}$$

$$= \frac{1}{k}(x_k + \sum_{j=1}^{k-1} x_j) \tag{5.2}$$

$$= \frac{1}{k}(x_k + (k-1)u_{k-1}) \tag{5.3}$$

$$= u_{k-1} + \frac{1}{k}(x_k - u_{k-1}) \tag{5.4}$$

The variable that we are averaging over is V(s) hence $V(s) = u_{k-1}$ and the counter $N(s) = k$ in equation 5.4. The return $G_t = x_k$. Hence at every time t:

$$N(S_t) = N(S_t) + 1 \tag{5.5}$$

$$V(S_t) = V(S_t) + \frac{1}{N(S_t)}(G_t - V(S_t)) \tag{5.6}$$

It can be useful to forget old states by making $\frac{1}{N(S_t)}$ a constant α as follows:

$$V(S_t) = V(S_t) + \alpha(G_t - V(S_t)) \tag{5.7}$$

Temporal Difference (TD) Learning

Temporal difference (TD) learning is a technique used to predict the value of a signals future state.

The aim of TD learning is to learn a value function v_π from experience of following policy π. The simplest form of TD is replacing the return G_t in equation 5.7 as follows:

$$V(S_t) = V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) \tag{5.8}$$

$[R_{t+1} + \gamma V(S_{t+1})]$ is known as the TD target, the value which $V(S_t)$ tends to as $t \rightarrow \infty$. $[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$ is known as the TD error. The aim of TD learning is to reduce the TD error to zero and update $V(S_t)$ to the estimated return $R_{t+1} + \gamma V(S_{t+1})$.

Mathematically we can show the relationship between MC and TD using the n-step return which is defined as:

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+1}) \tag{5.9}$$

For clarity we show $G_t^{(n)}$ for a few values of n as follows:

$$(TD)n = 1 : G_t^{(1)} = R_{t+1} + \gamma V(S_{t+1}) \tag{5.10}$$

$$n = 2 : G_t^{(2)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 V(S_{t+2}) \tag{5.11}$$

$$(MC)n = \infty : G_t^{(\infty)} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{T-1} R_T \tag{5.12}$$

Then n-step TD is defined as:

$$V(S_t) = V(S_t) + \alpha(G_t^{(n)} - V(S_t)) \tag{5.13}$$

What we now have is a function with which we can choose when to update V(S), instead of always having it update at the end of the episode like MC. This is useful because we can correct mistakes made in the prediction earlier if n-step TD is used.

5.2.2. Model-free control

When we previously used greedy policy improvement over the value function V(s) we had in equation 4.2:

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

Now for greedy policy improvement over the state action function Q(s,a) we have that:

$$\pi^{'}(s) = \max_{a \in A} Q(s, a) \tag{5.14}$$

Which means that we now maximize not only across the action set as in equation 4.2, but rather the entire state-action set when using model-free policy iteration. One should notice that to find the optimal policy we no longer need a model of the environment. In other words there is no dependency on the probability matrix $P_{ss'}^a$ as seen in equation 5.14.

However there now arises a problem when always having the agent act greedily. Which is that the agent will always take the same path no matter whether another path might have been better in the long term. To overcome this limitation we bring in the topic of ϵ-Greedy exploration.

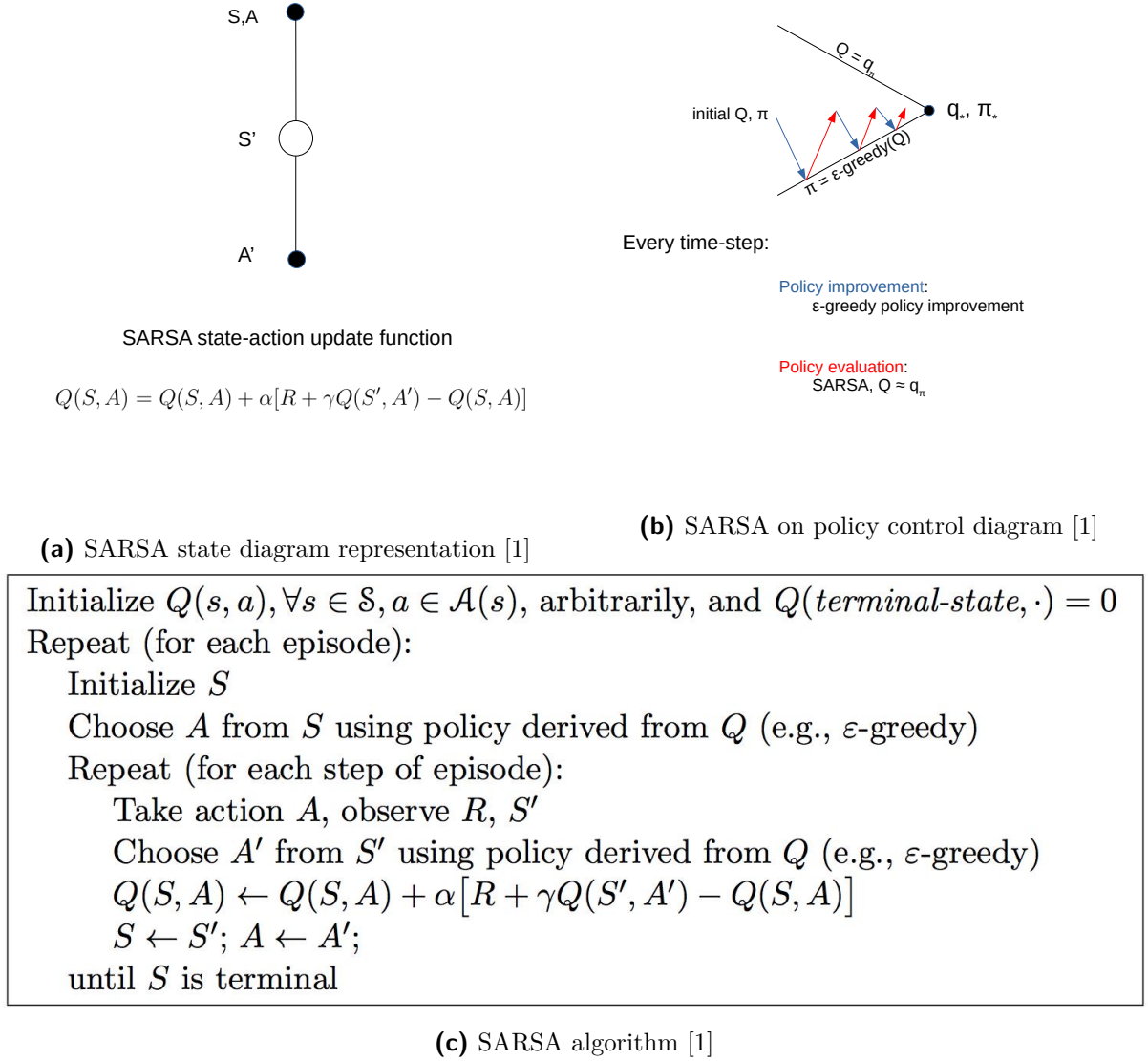To ensure that the agent explores occasionally instead of always taking the greedy action, we let the agent act randomly with a probability ϵ. This can mathematically be expressed as:

$$\pi(a|s) = \begin{cases} \frac{\epsilon}{m} + 1, & \text{if } a^* = \arg\max_{a \in A} Q(s, a) \\ \frac{\epsilon}{m}, & \text{otherwise} \end{cases} \tag{5.15}$$

One can note that equation 5.15 is a variation of equation 3.20.

SARSA

SARSA state-action update function

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

Every time-step:

Policy improvement:
ε-greedy policy improvement

Policy evaluation:
SARSA, $Q \approx q_\pi$

(a) SARSA state diagram representation [1]

(b) SARSA on policy control diagram [1]

Initialize $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(terminal\text{-}state, \cdot) = 0$
Repeat (for each episode):
    Initialize S
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Repeat (for each step of episode):
        Take action A, observe R, S'
        Choose A' from S' using policy derived from Q (e.g., ε-greedy)
        $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$
        $S \leftarrow S'; A \leftarrow A';$
    until S is terminal

(c) SARSA algorithm [1]

Figure 5.1: SARSA diagrams [1]

In Figure 5.1a the concept of the State-Action-Reward-State-Action is graphically shown. For every state action pair, we can calculate the q-function using the same concept as TD learning as follows:

$$Q(S, A) = Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)] \tag{5.16}$$

Where $R + \gamma Q(S', A')$ is the TD target. In Figure 5.1b we see that the way SARSA learns the optimal policy π_* is not by completely following policy π. But instead by stopping somewhere in between which can be seen by the arrows stopping in the middle of the two lines for Q and π. If Monte Carlo was instead used to learn the optimal policy π_* we would have a diagram similar to 4.1b, where the entire episode has to play out before the

16

algorithm updates. We can also define n-step Q-return:

$$q_t^{(n)} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n Q(S_{t+n}) \tag{5.17}$$

The goal of n-step SARSA is to update Q(s,a) towards the n-step Q-return $q_t^{(n)}$.

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[q_t^{(n)} - Q(S_t, A_t)] \tag{5.18}$$

When we have n = 1 then we have SARSA updates and at n = ∞ we have Monte Carlo updates. All values of n $\in [1, \infty]$ is between SARSA and Monte Carlo. Figure 5.1c shows the algorithm that can be used to implement SARSA. Later in chapter 7 we will implement this algorithm which will make the concept clearer.

Q-learning

(b) Q-learning algorithm [1]
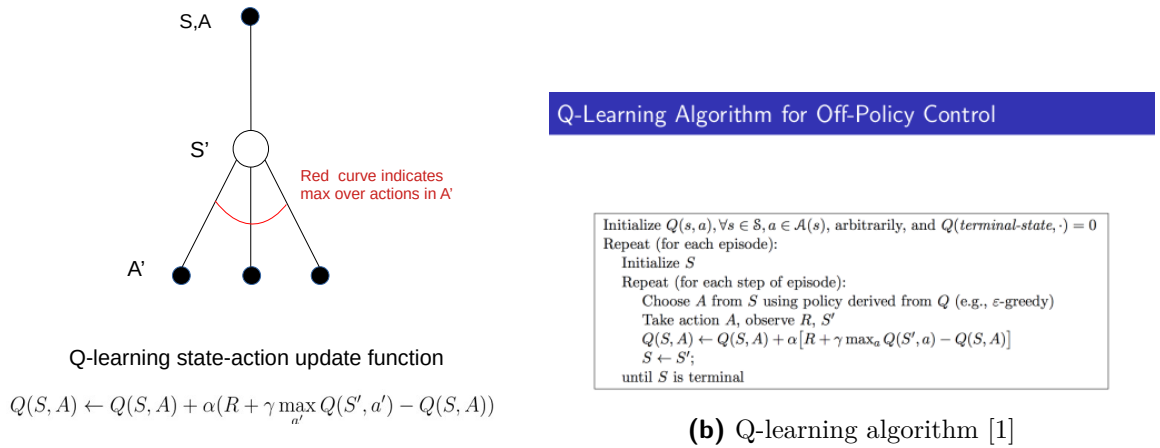
(a) Q-learning state action diagram [1]

Figure 5.2: Q-learning state-action diagram and algorithm

$$Q(S, A) \leftarrow Q(S, A) + \alpha(R + \gamma \max_{a'} Q(S', a') - Q(S, A)) \tag{5.19}$$

What we have been dealing with up to now in this chapter dealt with on-policy learning. We describe now Q-learning which falls under off policy learning. By comparing Figure 5.2a and Figure 5.1a, we can see that the only difference between Q-learning and SARSA is that Q-learning takes the maximum action A', instead of a singular determined A', which results in the maximum Q-return. Where the Q-return was earlier defined in equation 5.17. Hence in equation 5.19 we see the Q-learning update function is very similar to the

SARSA function in equation 5.16. Q-learning control always converges to the optimal action value function meaning, $Q(S, A) \rightarrow q_*(S, A)$.

Chapter 6

System Design

6.1. Introduction

The

    To solve the puzzle I could have used monte carlo learning, ... I chose SARSA and Q-learning because ...

6.2. Thing 2

6.3. Methods

- Measure reward-episodes graphs

Chapter 7

Applying reinforcement learning to sliding puzzles

7.1. Introduction

In the following section we will test the SARSA and Q-learning algorithms described in chapter 5. The way the tests will be conducted is by varying the hyper parameters. These hyper parameters include, the discount factor γ, epsilon the probability of choosing a random action and α the learning rate.

The puzzle that we will solve is of size 3x3. The way in which it will be solved is by training separate agents to solve different parts of the puzzle.

7.2. SARSA figs

7.3. Q-learning figs

7.4. Discussion

Chapter 8

Summary and Conclusion

8.1. Summary

Objective 1 ...

Objective 2 ...

Objective 3 ...

So what? Relates back to problem and background sections in introductions.

8.2. Future work and reccomendations

Bibliography

[1] D. Silver, "Ucl course on rl," 2015. [Online]. Available: https://www.davidsilver.uk/teaching/

[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] R. E. Korf, "Depth-first iterative-deepening: an optimal admissible tree search. artificial intelligence 27," pp. 97–109, (1985).

[5] R. S. Sutton and A. G. Barto, *Reinforcement Learning An Introduction second edition.* The MIT Press, 2018.

[6] H. M. Bastian Bischoff, Duy Nguyen-Tuong and A. Knoll, "Solving the 15-puzzle game using local value-iteration," 2013.

[7] P. Weng, "Markov decision processes with ordinal rewards: Reference point-based preferences," 2011.

[8] W. E. Johnson, Wm. Woolsey; Story, "Notes on the "15" puzzle," *American Journal of Mathematics*, vol. 29, no. 2(4), p. 397–404, 1879.

[9] 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

[10] Minsky, *Computation: finite and infinite machines.* Prentice Hall, 1967.

Appendix A

Solvability of a NxN sliding puzzle

info in this chapter to still be integrated somewhere else into report if needed. Else will leave it out

A.1. General puzzle description

Let us assume that we have an NxN puzzle, then we have NxN number of blocks. We can represent the puzzle as an NxN array, then we stack the array into a one dimensional array of 1 x (N*N). For example see the 4x4 puzzle in Figure A.1 we have a 1 x 16 array as: Array = (12,7,8,13,4,9,2,11,3,6,15,14,5,1,10). Before we describe the conditions for a sliding puzzle to be solvable, we first define the term "inversion". Assuming the the first index of the 1xN 2 array starts at the left top corner (valued 12) in Figure A.1, and that it runs from [0,(N*N)-1]. Then an inversion occurs when Array[index] ¿ Array[index+1] where index is an arbitrary integer between 0 and N*N-1. Hence in Figure A.1 we have a total: sum of inversions(Array) = 11 + 6 + 6 + 8 + 3 + 5 + 1 + 5 + 1 + 2 + 4 + 3 + 1 + 0 = 56.


Figure A.1: Example of a sliding puzzle

A.2. Conditions for solvability

Even and odd sized boards are analysed separately (where size = N).

For odd sized boards where N is odd we have the puzzle only being solvable if and only if the boards has an even number of inversions. The proof for this can be deduced by looking at Figure 2 and noting that for every switch of the blank block we have an even change in the sum of inversions of the board. [2]
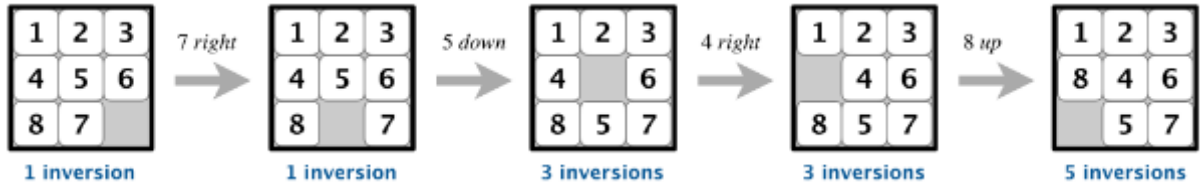
Figure A.2: Odd boards with change in blank piece only having even inversion change [2]

For even sized boards 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. This is illustrated in Figure 3.
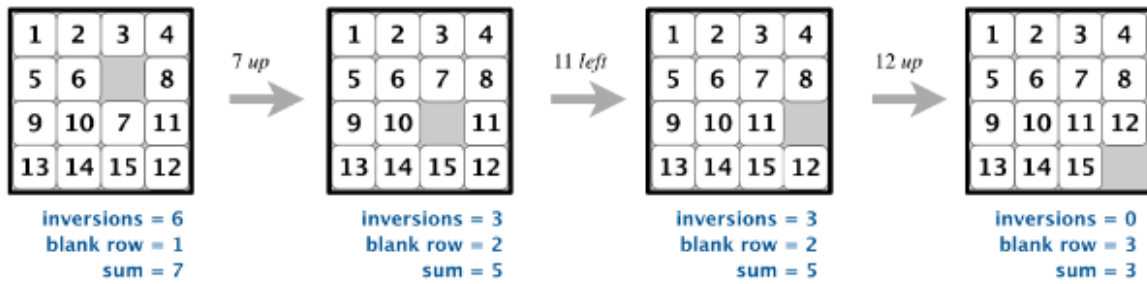
Figure A.3: Even board solvability [2]

Half of all puzzle configurations are unsolvable. [8] This means that we only have N! / 2 configurations that are solvable for an NxN board. This was proven using parity in the paper in [8]. Sliding puzzles can be solved relatively quickly with today's processing of computers for puzzles for example an 5x5 puzzle was solved in 205 tile moves in 2016. [9]

The issue more so lies in finding the shortest path to solving a puzzle. This specific problem of solving with the least amount of tile moves of a sliding puzzle has been defined as NP (non-deterministic polynomial-time) hard. NP hardness is are problems that are as least as hard as NP. Where in computational complexity theory NP (non-deterministic polynomial-time) is a has a solution with a proof variable to be in polynomial time by a deterministic Turing Machine. A Turing machine is a mathematical model defining an abstract machine which manipulates symbols according to a set of rules. [10]

In simpler terms a problem is NP if it can be solved within a time that is a polynomial function of the input. For instance if we define the time to solve a problem as 'T' and the input data as 'D'. Then as long as T = polynomial function (D) then a problem is NP.

Appendix B

Project Planning Schedule

This is an appendix.

Appendix C

Outcomes Compliance

This is another appendix.

Appendix D

Student and Supervisor agreement

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.

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.
2. It is the responsibility of the study leader to give regular guidance and feedback with regard to the literature, methodology and progress.
3. The rules regarding handing in and evaluation of the project is outlined in the Study Guide/website and will be strictly adhered to.
4. The project leader conveyed the departmental view on plagiarism to the student, and the student acknowledges the seriousness of such an offence.

Signature — study leader:	
Signature — student:	
Date:	5 August 2020

(Upload this form to SUNLearn)

Figure D.1: Student and Supervisor agreement