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

October 26, 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.

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 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 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: [illegible]
Initials and surname / *U.Barends*	Datum / *October 26, 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

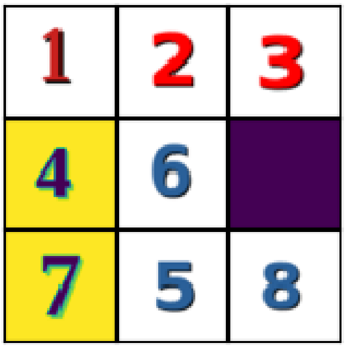
In robotics, a manipulator (ie. a robotic arm) is a device used to manipulate objects in its environment. When the manipulation task is relatively complex, it is often difficult or even impossible for a human to direct how the robot should act. One possible solution is that the robot learn by itself 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.

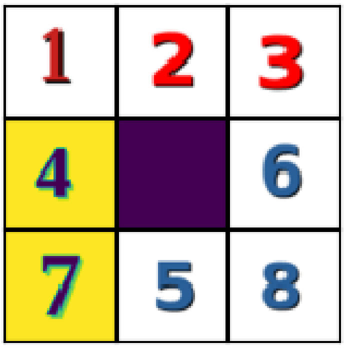
1.2. Background

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 $A_s = up, down, left, right$.

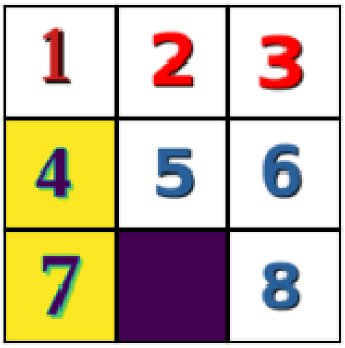
The sliding puzzle is a game with a history dating back to the 1870's [3]. Many solutions have been thought up to solve such puzzles, which include using search algorithms [4]. The rules of the game are that the only tiles that can move are the ones next to the tile with no number, known as the 'blank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



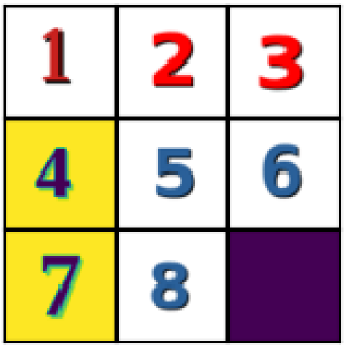
(a) Puzzle with tiles 5,6,8 and blank in the incorrect positions



(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 4 discusses in detail the dynamic programming and reinforcement learning techniques which will be used to solve our puzzle problem.

In Chapter 5 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 6 contains tests and verification that our solution works using the results of various experiments. Chapter 7 concludes the report.

Chapter 2

Literature Review/Study/Related work

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.

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 4. 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 10Î3 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

To model reinforcement learning problems we use dynamic systems theory, specifically incompletely-known MDP's (Markov decision processes). In general RL problems can be described using MDP's. In Mathematics a MDP is a stochastic, discrete time controlled process.

What this means is that a MDP is partially controlled and partially random. MDP's are dependent on the following variables: states, actions, state transition probability, reward and a discount factor [5]. These variables can be denoted in a 5-tuple as:

$$(S, A, P^a_{ss'}, R^a_s, \gamma)$$

where

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

For a state to be Markov it needs to satisfy the following condition:

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

In words this reads as, 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. For a MDP all states must be Markov.

We now define the definition which is the total discounted reward from time-step t, named the return G_t where:

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... && (3.1) \\ &= \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} && (3.2) \end{aligned}$$

The discount $\gamma \in$ [0,1] determines the present value of future rewards. For $\gamma = 0$ the return G_t only depends on the current reward that can be obtained in the next step R_{t+1}. Which can be said to be "short-sighted". While for $\gamma = 1$ the return depends on all the rewards that are projected to be obtained until the process terminates. This can be said to be "far-sighted". The larger γ is, the more the rewards of later steps closer to the terminating state affects the return. [5] What we can also note is that using a discount factor makes the return finite because R_{t+k+1} is finite and:

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

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

We now define another important definition, namely the state value function v(s) where:

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

Which in words means that the state value function v(s) is the expected return given that the agent is in state s at time-step t. We now decompose v(s) so that it becomes a recursive function as follows:

$$\begin{aligned} v(s) &= E[G_t | S_t = s] && (3.3) \\ &= E[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... | S_t = s] && (3.4) \\ &= E[R_{t+1} + \gamma (R_{t+2} + \gamma R_{t+3} + ...) | S_t = s] && (3.5) \\ &= E[R_{t+1} + \gamma G_{t+1} | S_t = s] && (3.6) \\ &= E[R_{t+1} + \gamma v(S_{t+1}) | S_t = s] && (3.7) \end{aligned}$$

This means that v(s) only depends on R_{t+1} the reward it can incur in the next step and $\gamma v(S_{t+1})$ the discounted state function at time t+1. What we have defined in equation 3.7 is known as the Bellman Equation. [5]

The probability an agent takes a certain of action given it is in a given state is defined as a policy $\pi(a|s)$. It is defined as:

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

Then we can now define the state value function following policy π as:

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

Additionally we define the action-value function also known as a q-function as:

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

The q-function is defined as the expected return taking action a from state s 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. While q(s,a) is the value for being in a certain state and taking a action, which has to do with control compared to v(s) which has to do with planning.

Once again we can decompose the state value and also action value functions to be in the form of the Bellman Equation in equation 3.7. This results in the Bellman *Expectation* equations:

$$\begin{aligned} v_\pi(s) &= E_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s] && (3.8) \\ &= \sum_{a' \in A} \pi(a|s)(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_\pi(s')) && (3.9) \end{aligned}$$

$$\begin{aligned} q_\pi(s,a) &= E_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a] && (3.10) \\ &= R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \sum_{a' \in A} \pi(a'|s') q_\pi(s', a') && (3.11) \end{aligned}$$

The Bellman Expectation equations can also then be put into matrix form from equation 3.9 for vector calculation as:

$$v_\pi = R^\pi + \gamma P^\pi v_\pi \tag{3.12}$$

Which has the solution:

$$v_\pi = (1 - \gamma P^\pi)^{-1} R^\pi \tag{3.13}$$

Equation 3.13 is a fast and concise formula which is easily implemented in code, but has the drawback of only working for small state spaces. This is due to the fact that the calculation requires the entire transition matrix P^π to be loaded into memory at every calculation step.

We now define what is known as the optimal policy, which is essentially a set of state

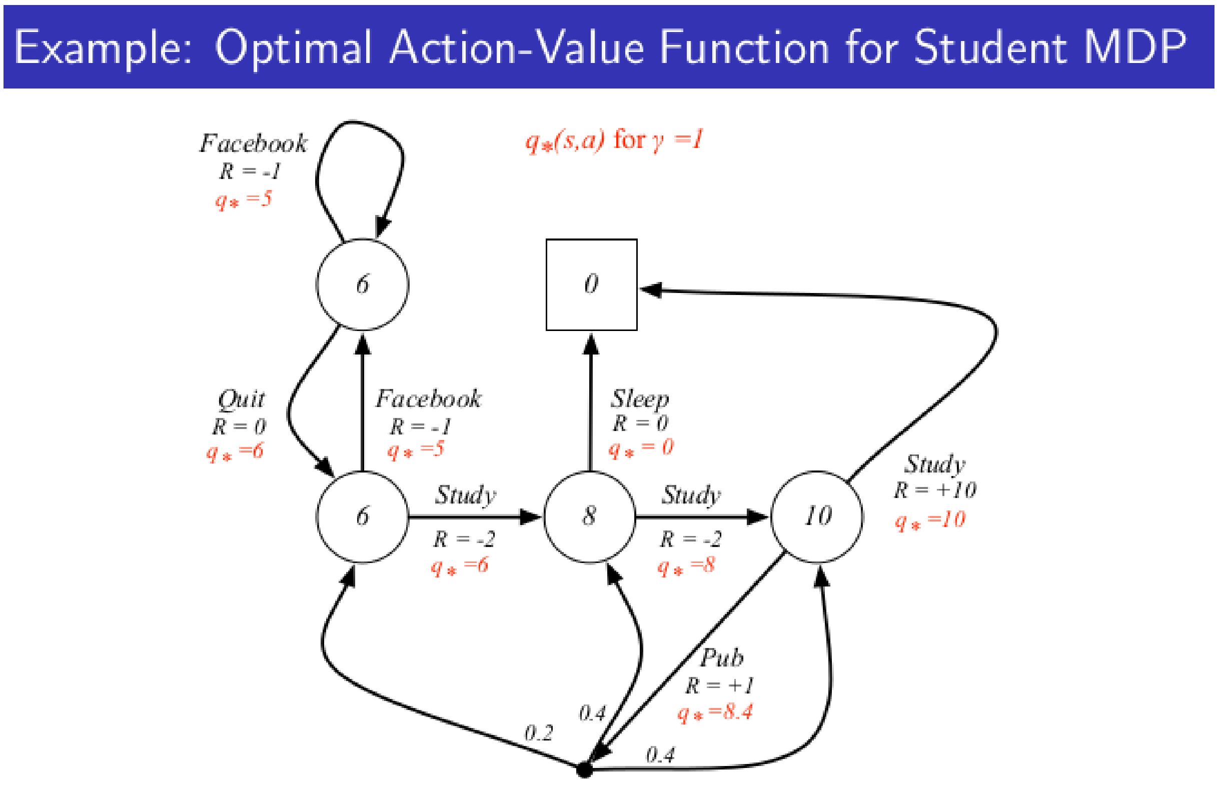


Figure 3.1: Graph of an example MDP showing how q values work [1]

and action pairs which lead to the maximum $q_\pi(s,a)$ as:

$$\pi_*(a|s) = \begin{cases} 1, & \text{if a} = \underset{a \in A}{\arg\max}\, q_*(s,a) \\ 0, & \text{otherwise} \end{cases} \tag{3.14}$$

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 to us how the agent must move to maximize its rewards, which is precisely the goal of RL. Figure 3.1 shows an example MDP displaying how $q_*(s,a)$ works. The red path lines indicate the optimal policy in Figure 3.1.

We now finally define what is know as the Bellman *Optimality* Equations which is a specific form of equation 3.7 and 3.9 which are:

$$v_*(s) = \max_a (R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_*(s')) \tag{3.15}$$

$$q_*(s,a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \max_{a'} q_*(s',a') \tag{3.16}$$

What equation 3.15 represents is the maximum reward that can be obtained in state s. While equation 3.16 represents the maximum reward that can be obtained by taking action a from state s. The difference between the two is that equation 3.15 is a passive prediction of what value (according to reward that can be obtained) a state represents. While equation 3.16 is a calculation of the value of being in a state after an action 'a' is selected.

Chapter 4

Dynamic Programming and Reinforcement Learning

4.1. Introduction

This chapter can be divided into three sections. Dynamic programming which **solves** a ***known*** MDP, model-free prediction which **estimates** an ***unknown*** MDP and model-free control which **optimizes** the value function of an ***unknown*** MDP.

4.2. Dynamic Programming

4.2.1. Background

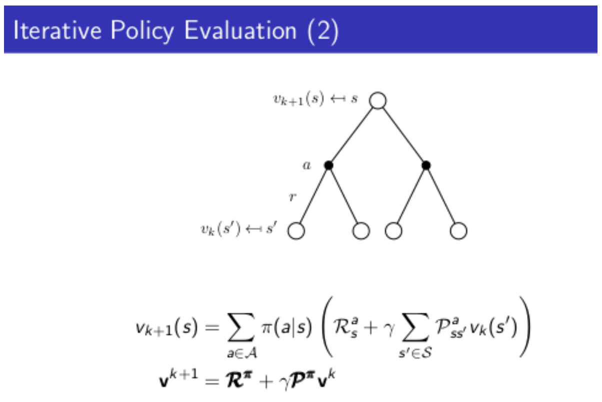
Dynamic programming is a method to solve problems by using a divide and conquer approach. That is to say a problem is broken into smaller sub-problems and then solved. There are two properties that must be met for a problem to be solvable via Dynamic Programming: Optimal substructure and Overlapping sub-problems. [1]

- Optimal substructure requires that the solution be able to be decomposed into sub-problems.
- Overlapping sub-problems requires that the sub-problems occur many times over, meaning 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-problems condition. [1]

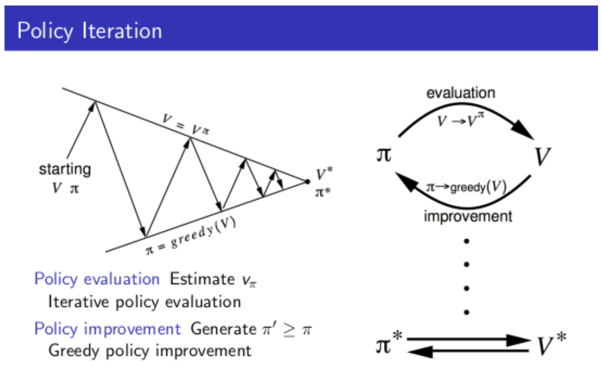
4.2.2. Iterative Policy Evaluation

In order to evaluate the value that a certain policy has, we use a method called *Iterative Policy Evaluation*. Figure 4.1a shows the mathematics which describes Iterative Policy


Iterative Policy Evaluation (2)

$$v_{k+1}(s) = \sum_{a\in\mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right)$$
$$\mathbf{v}^{k+1} = \mathcal{R}^{\pi} + \gamma \mathcal{P}^{\pi} \mathbf{v}^k$$

(a) Iterative policy Evaluation [1]


Policy Iteration

Policy evaluation Estimate v_π
Iterative policy evaluation
Policy improvement Generate $\pi' \geq \pi$
Greedy policy improvement

(b) Greedy Policy iteration visualisation [1]

Figure 4.1: Policy Iteration [1]

Evaluation in both summation and vector form. The diagram at the top of Figure 4.1a represents the value function being calculated for an action 'a' to an intermediary state represented by the black dot. The environments reaction is then also accounted for from the intermediary state to state s' resulting in reward r. The environments information is contained in the transition probability matrix P^{π}.

We then use state s' as the new start state s and repeat the same calculation. The value function is updated continuously until the difference between v^k and v^{k+1} is determined negligible. This process is known as Iterative Policy Evaluation.

4.2.3. 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.
- The control method only takes (S, A, P, R, γ) as input and outputs the optimal value function v_* and optimal policy π_*.

These two methods of Dynamic programming are implemented via what is known as value iteration and policy iteration respectively. Figure 4.1b shows how Greedy Policy Iteration works. A greedy policy is one which when followed, results in the maximum value function v_π. It is known that policy iteration converges which is represented in the diagram in Figure 4.1b on the top left. [5]

As is stated in Figure 4.1b policy iteration works by continuously iterating between two steps, namely policy evaluation and policy improvement. For a given policy π we improve the policy using the following two steps:

Step one is *Evaluating* the policy π using equation 3.9:

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

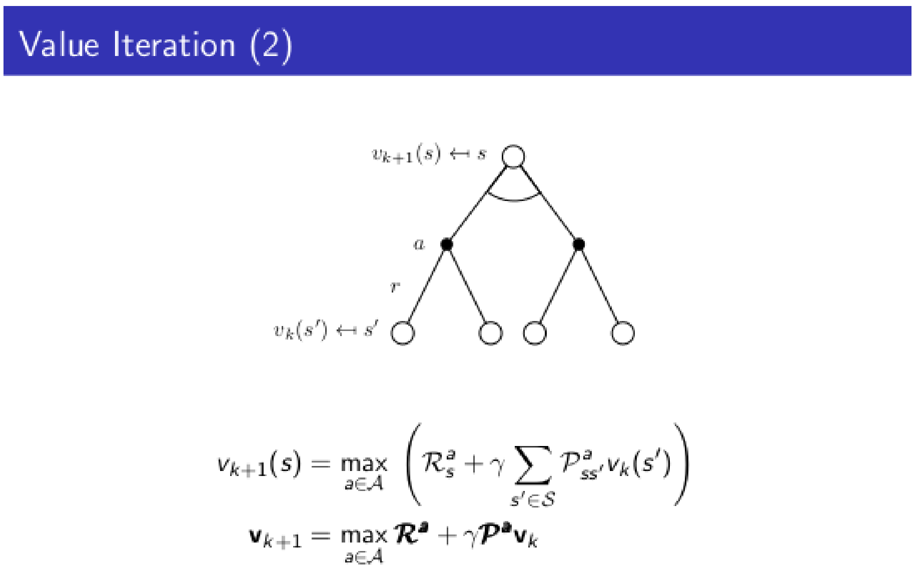
Step two is *Improving* the policy by acting greedy with respect to v_π to obtain a new policy $\pi^{'}$:

$$\pi^{'} = greedy(v_\pi) = v_* \tag{4.1}$$

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

What this means is that we obtain a value function by using iterative policy evaluation where after we update the policy. In this way we will always converge to the optimal policy π_* and value function v_* described by equation 3.14 and 3.15 respectively.

4.2.4. Value Iteration



Value Iteration (2)

$$v_{k+1}(s) = \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right)$$

$$\mathbf{v}_{k+1} = \max_{a \in \mathcal{A}} \boldsymbol{\mathcal{R}}^{\mathbf{a}} + \gamma \boldsymbol{\mathcal{P}}^{\mathbf{a}} \mathbf{v}_k$$

Figure 4.2: Value Iteration [1]

Figure 4.2 shows the mathematics that describes *Value Iteration*. It is very similar to iterative policy evaluation in Figure 4.1a. Iterative Policy Evaluation calculates the value function for a policy π which is an entire set of actions given states.

While Value Iteration calculates the maximum value function for a single action only. In this sense Value Iteration is a special case of Iterative Policy Evaluation.

4.3. Reinforcement Learning

What we have been discussing up to now all has had to do with using methods which required knowledge about the model. 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

4.3.1. 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:

$$u_k = \frac{1}{k}\sum_{j=1}^{k} x_j \tag{4.2}$$

$$= \frac{1}{k}(x_k + \sum_{j=1}^{k-1} x_j) \tag{4.3}$$

$$= \frac{1}{k}(x_k + (k-1)u_{k-1}) \tag{4.4}$$

$$= u_{k-1} + \frac{1}{k}(x_k - u_{k-1}) \tag{4.5}$$

The variable that we are averaging over is V(s) hence $V(s) = u_{k-1}$ and the counter $N(s) = k$ in equation 4.5. The return $G_t = x_k$. Hence at every time t:

$$N(S_t) = N(S_t) + 1 \tag{4.6}$$

$$V(S_t) = V(S_t) + \frac{1}{N(S_t)}(G_t - V(S_t)) \tag{4.7}$$

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

$$V(S_t) = V(S_t) + \alpha(G_t - V(S_t)) \tag{4.8}$$

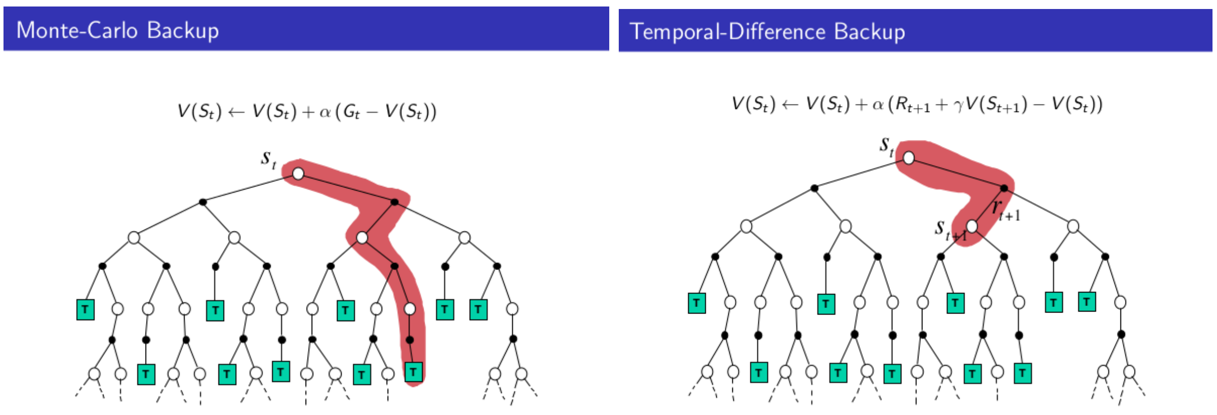
Temporal Difference (TD) Learning

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 4.8 as follows:

$$V(S_t) = V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) \tag{4.9}$$

$[R_{t+1} + \gamma V(S_{t+1})]$ is known as the TD target, the value which $V(S_t)$ tends to as $t \to \infty$. $[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$ 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 $R_{t+1} + \gamma V(S_{t+1})$.


Monte-Carlo Backup
$V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t))$
S_t
T
Temporal-Difference Backup
$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t))$
S_t
r_{t+1}
S_{t+1}
T

(a) Monte Carlo backup [1]

(b) Temporal difference backup [1]

Figure 4.3: MC and TD visual representations using n-return [1]

Figure 4.3a and 4.3b shows the concepts of MC and TD learning visually presented. Mathematically we can show the relationship between MC and TD using the n-step return concept, which will now be described.

Let the n-step return be defined as:

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+1}) \tag{4.10}$$

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

$$(TD) n = 1 : G_t^{(1)} = R_{t+1} + \gamma V(S_{t+1}) \tag{4.11}$$

$$n = 2 : G_t^{(2)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 V(S_{t+2}) \tag{4.12}$$

$$(MC) n = \infty : G_t^{(\infty)} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{T-1} R_T \tag{4.13}$$

Figure 4.3a corresponds to equations 4.8 and 4.11. While Figure 4.3b is related to equation 4.9 and 4.13. We now define n-step TD as:

$$V(S_t) = V(S_t) + \alpha(G_t^{(n)} - V(S_t)) \tag{4.14}$$

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.

4.3.2. Model-free control

When we previously used greedy policy improvement over the value function V(s) we had in equation 4.1:

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

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

$$\pi^{'}(s) = \max_{a \in A} Q(s, a) \tag{4.15}$$

Which means that we now maximize not only across the action set as in equation 4.1, 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 4.15.

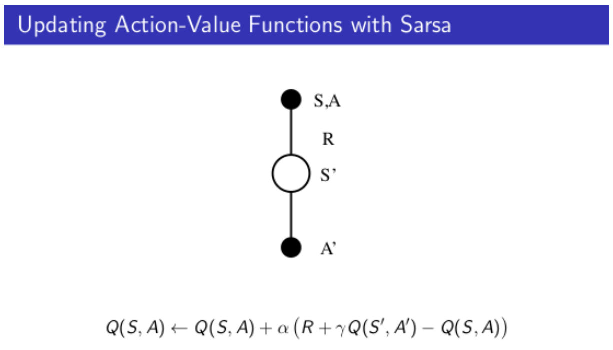
However there now arises a problem when always having the agent act greedily. Which is that the agent will always take the same path no matter whether another path might have been better in the long term. To overcome this limitation we bring in the topic of ϵ-Greedy exploration.

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:

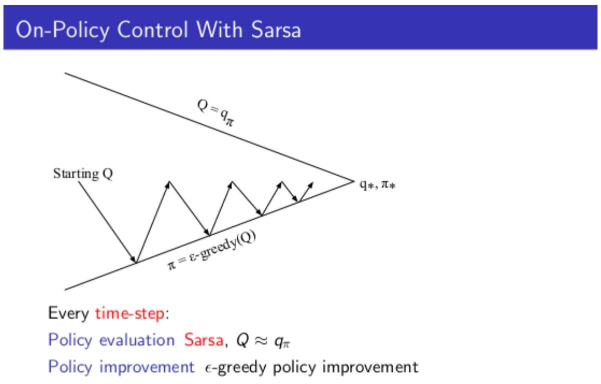
$$\pi(a|s) = \begin{cases} \frac{\epsilon}{m} + 1, & \text{if } a^* = \arg\max_{a \in A} Q(s, a) \\ \frac{\epsilon}{m}, & \text{otherwise} \end{cases} \tag{4.16}$$

One can note that equation 4.16 is a variation of equation 3.14.

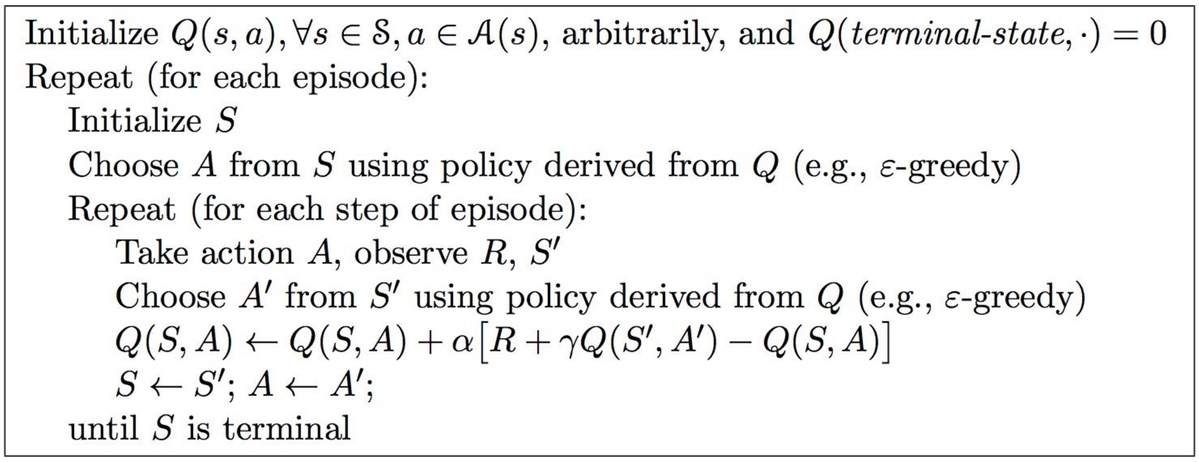
SARSA


Updating Action-Value Functions with Sarsa
S,A
R
S'
A'
$Q(S,A) \leftarrow Q(S,A) + \alpha\,(R + \gamma Q(S',A') - Q(S,A))$

(a) SARSA state diagram representation [1]


On-Policy Control With Sarsa
$Q = q_\pi$
Starting Q
q_*, π_*
$\pi = \epsilon\text{-greedy}(Q)$
Every time-step:
Policy evaluation Sarsa, $Q \approx q_\pi$
Policy improvement ϵ-greedy policy improvement

(b) SARSA on policy control diagram [1]



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

(c) SARSA algorithm [1]

Figure 4.4: SARSA diagrams [1]

In Figure 4.4a 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:

$$Q(S,A) = Q(S,A) + \alpha[R + \gamma Q(S',A') - Q(S,A)] \tag{4.17}$$

Where $R + \gamma Q(S',A')$ is the TD target. In Figure 4.4b 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 algorithm updates. We can also define n-step Q-return:

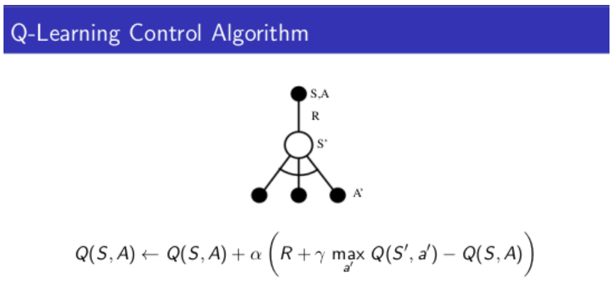
$$q_t^{(n)} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n Q(S_{t+n}) \tag{4.18}$$

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

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[q_t^{(n)} - Q(S_t, A_t)] \tag{4.19}$$

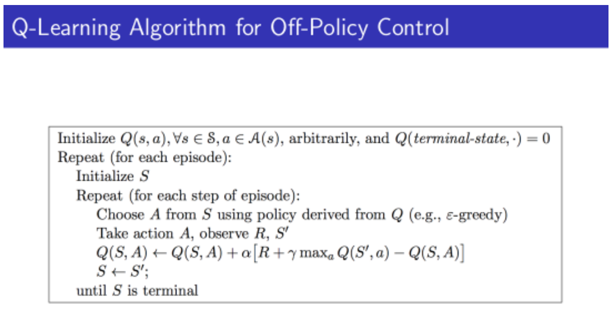
When we have n = 1 then we have SARSA updates and at n = ∞ we have Monte Carlo updates. All values of n $\in [1, \infty]$ is between SARSA and Monte Carlo. Figure 4.4c shows the algorithm that can be used to implement SARSA. Later in chapter 6 we will implement this algorithm which will make the concept clearer.

Q-learning



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

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



Q-Learning Algorithm for Off-Policy Control

```
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 $S$
    Repeat (for each step of episode):
        Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        Take action $A$, observe $R$, $S'$
        $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$
        $S \leftarrow S'$;
    until $S$ is terminal
```

(b) Q-learning algorithm [1]

Figure 4.5: Q-learning state-action diagram and algorithm

What we have been dealing with up to now in this chapter dealt with on-policy learning. We now move to describe Q-learning which falls under off policy learning. By comparing Figure 4.5a and Figure 4.4a, we can see that the only difference between Q-learning and SARSA is that Q-learning takes the action A' which gives the maximum Q-return. Where the Q-return was earlier defined in equation 4.18. Then in Figure 4.5b we see the Q-learning algorithm which can be seen to be very similar to the SARSA algorithm in Figure 4.4c.

Chapter 5

System Design

5.1. Introduction

The

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

5.2. Thing 2

5.3. Methods

- Measure reward-episodes graphs

Chapter 6

Applying reinforcement learning to sliding puzzles

6.1. Introduction

In the following section we will test the SARSA and Q-learning algorithms described in chapter 4. 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.

6.2. SARSA figs

6.3. Q-learning figs

6.4. Discussion

Chapter 7

Summary and Conclusion

7.1. Summary

Objective 1 ...

Objective 2 ...

Objective 3 ...

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

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

[8] 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

[9] 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.

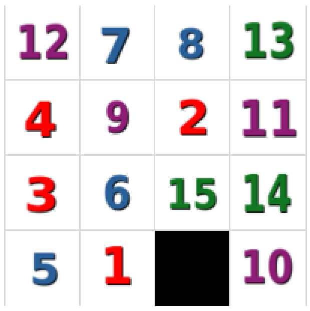


Figure A.1: Example of a sliding puzzle

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]

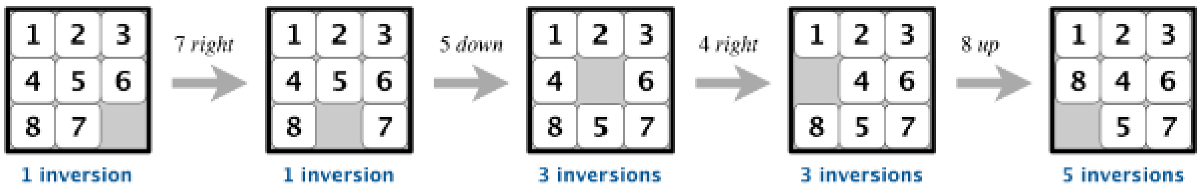


Figure A.2: Odd boards with change in blank piece only having even inversion change [2]

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.

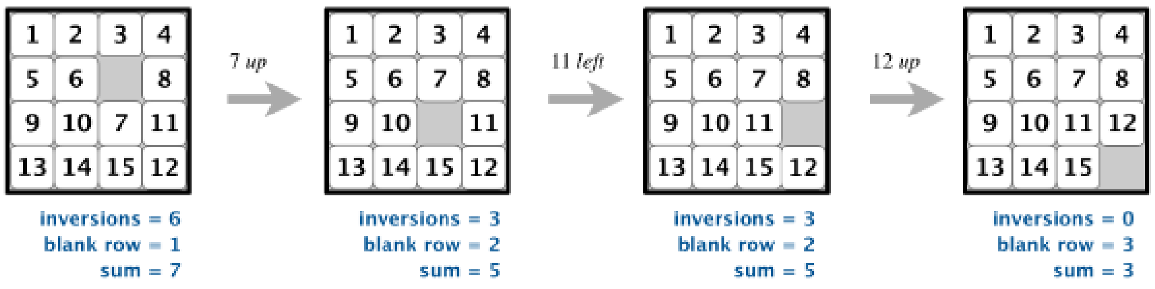


Figure A.3: Even board solvability [2]

Half of all puzzle configurations are unsolvable. [7] 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 [7]. 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. [8]

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. [9]

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.

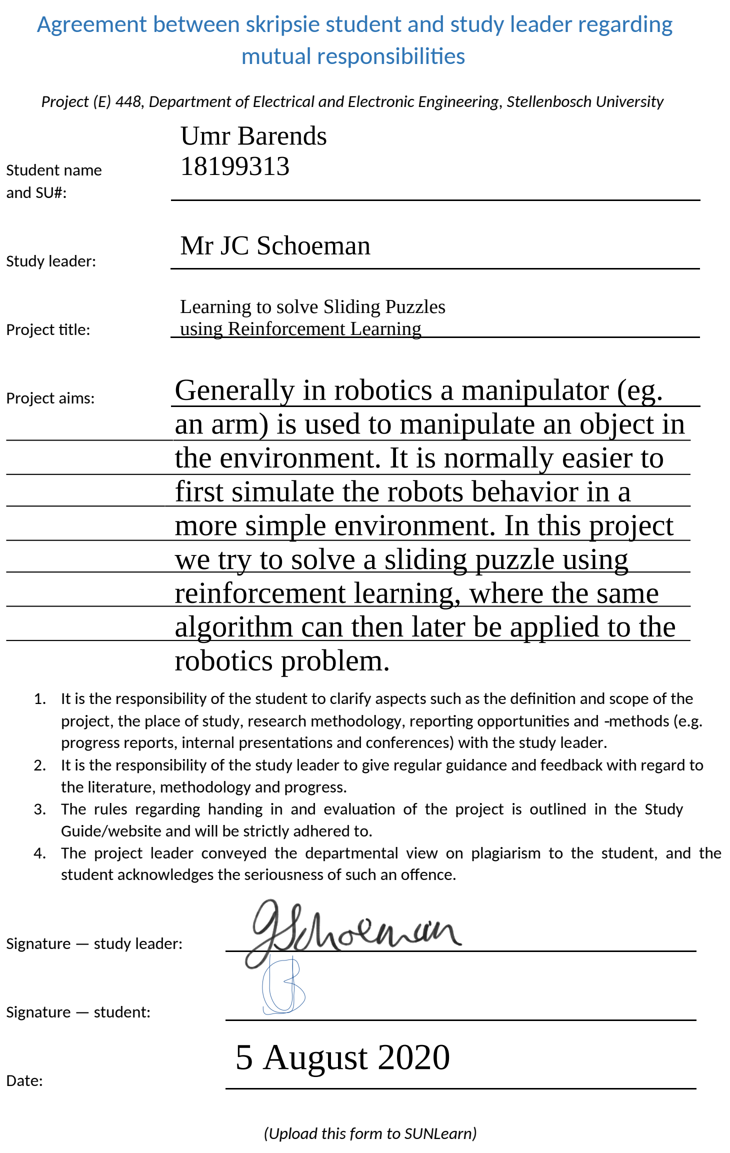
Appendix C

Outcomes Compliance

This is another appendix.

Appendix D

Student and Supervisor agreement



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.

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:	[illegible]
Signature — student:	[illegible]
Date:	5 August 2020

(Upload this form to SUNLearn)

Figure D.1: Student and Supervisor agreement