Reinforcement Learning for minimizing the total waiting time of passengers in a Taxi Dispatch Problem

Vera Martens
(4590546)

A thesis presented for obtaining the degree of
Bachelor of Science in Applied Mathematics

Optimization
Delft Institute of Applied Mathematics
Delft University of Technology
The Netherlands
January 2020

Supervisor:
Dr. ir. J.T. van Essen

Assessment Committee:
Dr. ir. J.T. van Essen
Drs. E.M. van Elderen
Dr. ir. W.G.M. Groenevelt

TU Delft Delft University of Technology

Abstract

A *Taxi Dispatch Problem* involves assigning taxis to requests of passengers who are waiting at different locations for a trip. In today's economy and society, the Taxi Dispatch Problem and other transport problems can be found everywhere. Not only in transporting people, but also in food delivery from restaurants and package delivery for all kind of companies. Even though the applications are different, they still have something in common: serving as much as requests as possible, because that means the highest income. In this thesis, we consider the problem in the actual taxi field. A taxi driver often chooses to serve the passenger that is closest, because he makes no money while the taxi is vacant. However, for companies such as Uber, this is probably not the best solution. They have an overview of the locations of the taxis and passengers, and therefore, are able to make an optimal assignment between the taxis and requests. Sometimes, waiting a little longer for new requests leads to an even better solution.

Trying to optimize the problem for the long-run and predict where passengers appear and where taxis end up is perfectly suited for *Reinforcement Learning* (RL), a subfield of Machine Learning. To be able to solve an optimization problem such as a Taxi Dispatch Problem, there needs to be a goal. For a company, this is maximizing the income or profit, and a popular way to do that is by minimizing the travel time. This thesis takes a different approach by looking at the problem from the passenger's perspective, as satisfied passengers lead to more passengers.
In this thesis, the goal is to find an optimal policy for assigning taxis to passengers such that the total waiting time over all passengers is minimized, by using Reinforcement Learning.

In order to do that, we formulate the problem in terms of the elements of an RL problem, with the RL method Q-Learning as the learning algorithm and ϵ-Greedy as policy. Together with some restrictions and assumptions, we implement this in Java and use this program to make the agent learn and generate results. The agent is the one that is responsible for assigning passengers to taxis and needs to learn how to make this assignment such that the total waiting time is minimized.

We say that the learning algorithm has converged to the optimal policy as soon as it keeps choosing the same best actions for each state. To check this, we run the program ten times and compare the best actions per state. If the best actions of the different runs are the same, this means that the learning algorithm has converged. We start with a simple version of the problem, from which we build to a version that is much more realistic. In this last version, we work in an environment in which passenger requests appear at random moments and the agent can choose to not serve a passenger right away.

Unfortunately, the learning algorithm does not fully converge for all states. It might be that it still converges when taking high values for the number of episodes and time steps, but so far these are not found. We still can make conclusions about the optimal policy and see that there seems to be a sort of pattern. One of the interesting occurrences is when there are two available taxis and two passengers. The best action is then always to serve only one passenger (the one that leads to the smallest waiting time) and let the other wait. The agent never chooses to serve

no passengers, even though, at that moment, this results in the smallest waiting time. Also, the policy does not choose to serve both passengers, and it let one passenger wait a little longer. The policy seems to take the future into account.

Since the algorithm does not fully converge when using Q-Learning, it might be better to use a different method, such as Model-Based Reinforcement Learning.

Contents

Chapter 1

Introduction

A *Taxi Dispatch Problem* involves assigning taxis to requests of passengers who are waiting at different locations for a trip (Alshamsi, 2009). In today's economy and society, the Taxi Dispatch Problem and other transport problems can be found everywhere. Not only in transporting people, but also in food delivery from restaurants and package delivery for all kind of companies (Kuo, 2016). Even though the applications might be different, they still have something in common: serving as much as requests as possible, because that means the highest income.

In the taxi field, a lot of decisions are made based on experience. Taxi drivers wait at locations where they had success in finding a passenger in the past. Also, a taxi driver often chooses to serve the passenger that is closest. This is because the taxi driver makes no money while the taxi is vacant. If the taxi would drive to a passenger further away, the vacant driving time would be longer. However, for a company that has an overview of the locations of the taxis and passengers, such as Uber, assigning taxis to the closest passenger is probably not the best solution.

Consider the situation in Figure 4.1. There are two taxis T_1 and T_2, and two passengers P_1 and P_2 who have sent in a request for a trip. The driving time between any two locations is given. Suppose that taxi T_1 would serve passenger P_2 because P_2 is the closest, namely only 5 minutes away. Then, taxi T_2 serves passenger P_1, who is 25 minutes away. Now, the total driving time of the taxis is $5 + 25 = 30$ minutes. However, if taxi T_1 is assigned to passenger P_1 and taxi T_2 to passenger P_2, the total driving time is $10 + 15 = 25$ minutes. The total driving time of this second assignment is smaller than the one of the first assignment. This makes the second assignment therefore better, since the taxis lose less valuable time.

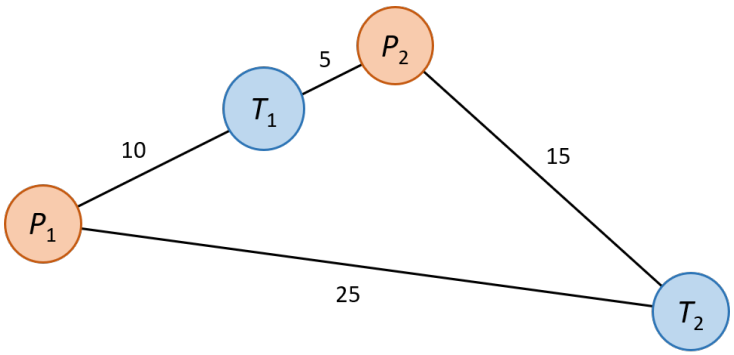

Figure 1.1: A graphical representation of a situation with two taxis T_1 and T_2 and two passengers P_1 and P_2, with the driving times in minutes given on the edges.

Of course, for the situation above, it is not that hard to see what the best solution is. However,

companies are working with a large number of taxis and requests, and therefore, they can use algorithms to solve the problem for them. The algorithm used in the example above is called *Route Optimization* (Kuo, 2016).

Even though this algorithm works good to find a solution for every individual time step, another thing to keep in mind is what is good in a long-run. Where taxis end up after dropping off their passengers, which is a result of an assignment, might not be optimal to the locations of the new passengers. This problem of predicting where passengers will show up and where taxis will end up after an assignment is a problem perfectly suited for *Machine Learning* (Zander, 2017). In this thesis, we use *Reinforcement Learning*, a subfield of Machine Learning. Section 2.1 explains why.

Whatever algorithm or method is used to solve a Taxi Dispatch Problem, there always needs to be a goal. For example, a company could want to maximize the income or profit, which can be done by minimizing the total driving time, as we have seen in the example on the previous page.
A different approach is to look at the problem from the passenger's perspective, because satisfied passengers lead to more passengers. Therefore, in this thesis, the goal is to find an optimal policy for assigning taxis to passengers such that the total waiting time over all passengers is minimized.

1.1 Related work

Quite some research has been done within the taxi world by using Reinforcement Learning. Many times this is done with the goal to maximize the revenue of a taxi (company). Verma *et al.* (2017) used Reinforcement Learning to learn an agent to maximize the earned revenue from the driver's perspective by using current trips of taxis and trips from the past. They used data from Singapore for the learning process. Wang & Lampert (2014) applied Reinforcement Learning to New York City taxi data to learn an agent how to maximize the revenue generated by a single taxi driver. In both cases, historical data was used. In this thesis, we do not make any use of data, although the problem could also be solved by the use of data.
Dietterich (1999) presents in his paper a learning algorithm that does not need data either. The algorithm is used to learn a taxi how to navigate to a passenger with the shortest route possible and to pick up and drop off the passenger on the right locations. Since there is always only one passenger present in the environment, the goal could also be formulated as minimizing the waiting time of the passenger. However, in this thesis, we learn the agent to do this for multiple passengers.
Alshamsi *et al.* (2009) solved a Taxi Dispatch Problem with a multi-agent system. They kept the waiting time of the passengers in mind as the passenger who was waiting the longest would get assigned first. However, priority is set on taxis that are vacant the longest rather than minimizing the total waiting time of the passengers.
Crites and Barto (1998) use multi-agent Reinforcement Learning for a similar problem: Elevator Group Control. In this problem, passengers are served by multiple elevators such that the total waiting time of the passengers is minimized. This problem could be seen as a Taxi Dispatch Problem when replacing the elevators by taxis. However, it is not known in advance which level the passenger wants to go to. In this thesis, we assume that the destination of the passenger is known in advance.

1.2 Organisation of thesis

This section describes how this report is organised and what each chapter is about.

Chapter 2 explains what Reinforcement Learning is and introduces all the concepts that are needed to understand the remainder of the thesis. It gives a background, discusses all the basic elements of a Reinforcement Learning problem and presents two algorithms that can be used in the learning process. In Chapter 3, the problem of this project is stated. It explains what a Taxi Dispatch Problem is, gives a description and formulation of the project's problem and lists all the restrictions and assumptions that are made during the project which form three different versions of the problem. Chapter 4 explains how the experiment is set up, how we make use of a Java program to make the agent learn, defines the convergence of the learning algorithm by two different outputs and analyzes these results for each different version of the program. Chapter 5 summarizes the conclusions and results. Chapter 6 discusses why it might have been better to solve the project's problem with model-based Reinforcement Learning as the learning algorithm does not fully converge in the final version of the program.

Chapter 2

Reinforcement Learning

This chapter explains what Reinforcement Learning (RL) is. Section 2.1 gives an overview of Artificial Intelligence, Machine Learning and Reinforcement Learning and how they are related to each other. Section 2.2 gives a better understanding of RL by explaining Markov Decision Processes and the basic elements of an RL problem. Section 2.3 presents two algorithms that can be used for value updating and explains what the difference is. Section 2.4 discusses the balance between exploration and exploitation. The content of the last three sections is based on Sutton & Brato (2018), and Güldenring (2019). At last, Section 2.5 gives an example of an RL problem and shows the difference between SARSA and Q-Learning.

2.1 Background

Artificial Intelligence (AI) is an area of computer science that encompasses creating intelligent computers and machines that can imitate human behaviour, thinking and reasoning, such as solving problems by itself. In the case of a computer, solving problems by itself is learning and improving from experience without being explicitly told how or any other human help.
Machine Learning (ML) is an area of Artificial Intelligence that focuses on algorithms that use data to learn. The process of learning is done by adapting to new circumstances and finding patterns in the provided data to make better decisions in the future (Russell & Norvig, 2010).
Machine Learning has three main subfields: Supervised Learning, Unsupervised Learning and Reinforcement Learning.

In Supervised Learning, an algorithm learns from a training data set, which consists of labeled input-output pairs. The system tries to predict the output for a certain input and compares its prediction with the correct answer to get feedback on how accurate it is. After sufficient training, the algorithm should give the right output for new input data. An example of Supervised Learning is classification.

In Unsupervised Learning, an algorithm tries to find a pattern or structure in given data that does not contain any labels. An example of Unsupervised Learning is clustering. The system tries to look for similarities in the data and uses that to cluster the inputs.

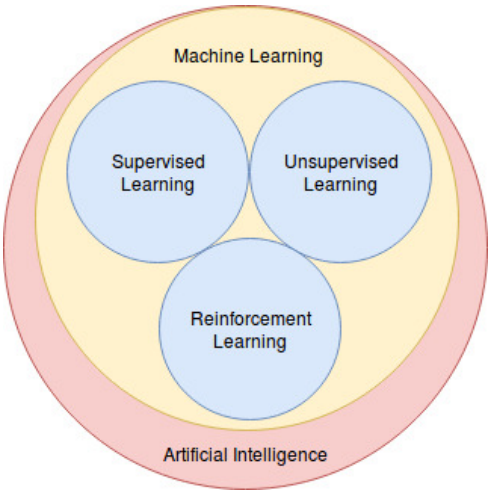

Figure 2.1: Schema of the subfields of Artificial Intelligence (Sharma, 2018).

Reinforcement Learning (RL) has an agent, for example a robot, that learns how to behave

within a dynamic environment by taking actions and getting a (possibly negative) reward for each action. The reward is a feedback for the agent to know whether the action was good. This helps the agent to attain its goal. Even though the agent receives feedback, Reinforcement Learning is different from Supervised Learning, since it is not known what the correct action would have been. However, Reinforcement Learning is also different from Unsupervised Learning, where there is no explicit feedback at all (Buşoniu *et al.*, 2010; Sutton & Barto, 2018). This is why RL is a separate subfield within Machine Learning.

There is also a challenge that arises in Reinforcement Learning and not in the other subfields: finding a balance between exploration and exploitation. On the one hand, the agent needs to use its experience to choose actions that are found to be good, since it wants to attain its goal as good as possible. This is called *exploitation*. However, the agent does not have any experience without discovering which actions *are* good, so it also needs to try new actions. This is called *exploration*. The agent needs both to achieve its goal. In Section 2.4, we discuss ways to find a balance between exploration and exploitation.

2.2 Markov Decision Processes

Markov Decision Processes (MDPs) frame the problem of learning to attain a goal from interaction. An RL problem can almost always be formalized as an MDP (Van Otterlo & Wiering, 2012). Understanding MDPs makes it easier to understand Reinforcement Learning. This section explains the main elements of MDPs and RL, such as the environment, rewards and policy.

2.2.1 Agent and environment

As well in Reinforcement Learning as in Markov Decision Processes, there is an agent that interacts with an environment. The *agent* is the one that makes decisions and learns from them. Everything outside the agent is called the *environment*. The agent and environment interact with each other over a sequence of discrete time steps $t = 0, 1, 2, \ldots$. At each time step t, the agent receives a representation of the environment at that moment, called the *state* s_t. Based on that state, the agent decides what *action* a_t to select from the possible actions. This brings the environment in a new state s_{t+1} and the agent receives, in time step $t + 1$, its *reward* r_{t+1}. Repeating this, gives the following sequence: $s_0, a_0, r_1, s_1, a_1, r_2, \ldots$. Figure 2.2 gives a schematic representation.

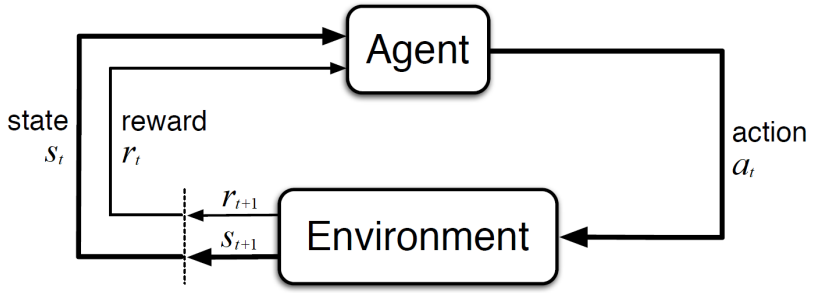

Figure 2.2: A schematic representation of the interaction between the agent and the environment. The agent selects an action based on the current state and received reward. The environment returns the new state and the reward based on the selected action (Sutton & Barto, 2018).

What the next state will be, is decided by the *state transition function* $T(s, a, s') = P(s'|s, a)$. This is the probability of ending up in state s' from state s after taking action a.

2.2.2 Goals, rewards and expected return

The goal of the agent is formalized in terms of the reward. To train an effective agent, the goal should be to maximize the total amount of reward received. That is, maximizing the reward in the long run instead of the immediate reward. Therefore, we seek to maximize the *expected return* G_t, which is in the simplest case equal to the sum of the rewards since time step t:

$$G_t = r_{t+1} + r_{t+2} + \ldots + r_T, \tag{2.1}$$

where T is the final time step.

To better understand why, consider the environment of Figure 2.3. The agent is in room one and needs to end up in room five. The reward is the negative distance between the agent and room five. So for example, when the agent is in room one, the reward is -2, since the distance between room one and room five is 2. If the agent would try to maximize the immediate reward, it would go to room three. The reward is then -1, which is higher than the reward of going to room two, which is -3 or the reward of staying in room one, which is -2. However, the agent will now always stay in room three, since it maximizes the immediate reward, and thus it will never reach room five. If the agent would try to maximize the expected return, it would got to room two, and then room four, six and five. It now reached the goal and maximized the expected return. Namely, when the agent goes directly to room 2 from its start position, the expected return is $G_{t=0} = r_1 + r_2 + r_3 + r_4 = -3 - 2 - 1 + 0 = -6$. If the agent had first gone to room three, the rewards would have been better, but after a while staying in room three the expected return would be worse, since the agent did not reach its goal yet. So the expected return would become $G_{t=0} = r_1 + r_2 + \ldots + r_6 + \ldots = -1 - 1 - \ldots - 1 - \ldots = -6 - \ldots$. So for the agent to reach its goal, it still has to leave room three and eventually go to room two.

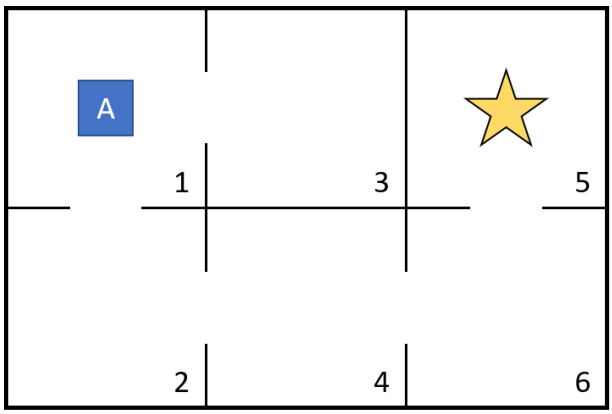

Figure 2.3: An environment with the agent in room one and the end goal in room five. The agent needs to maximize the expected return rather than the intermediate reward to reach its goal (Güldenring, 2019).

When the rewards that will be received in the future are less important than the immediate reward, the agent can try to select actions to maximize the *discounted expected return*:

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}, \tag{2.2}$$

where $\gamma \in [0, 1]$ is a parameter called the *discount rate*. When $\gamma < 1$, it makes a future reward worth less than what it would be worth if it was received immediately. In time step t, the reward received in time step $t + k$ is worth only γ^{k+1} times its original value. If $\gamma = 0$, then only the immediate reward remains, and thus, $G_t = r_{t+1}$. If $\gamma = 1$, all future rewards are considered of

same importance. Now the discounted expected reward is equal to (2.1).

The (discounted) expected return makes sense in problems that have a natural final time step T. This is when the agent reaches a *terminal state*. The period of time steps from the beginning till the final time step is called an *episode*. A terminal state resets the scene and brings the agent back to its start state. Now the next episode can start. This way, the training process consists of multiple episodes. On the other hand, in many cases, the problem does not naturally break into episodes, but is a continuous ongoing problem. That means that $T = \infty$, and thus the expected return G_t that we try to maximize is also infinite. Only if we truncate the series when γ^k is smaller than a given value, the sum is finite.

It looks like that the rewards must be provided in such a way that when maximizing them or the expected return, the agent achieves its goal. However, the reward is not used to let the agent know how to reach its goal. For this, it is better to use the policy and the value function.

2.2.3 Policy and value function

A *policy* tells the agent how to behave in the environment at a certain time step. It decides what action should be taken given the current state of the environment. Formally, the policy π is a probability distribution over the number of possible actions for each state. Then, $\pi(a|s)$ is the probability that action a is selected when the current state is s.

A *value function* is a function from a state to a *value* that specifies how good it is for the agent to be in a certain state and how good this is in the long-run. The value function of state s under policy π, denoted by $V_\pi(s)$, is the expected return when the start state is s and the agent uses policy π to select actions. Then, $V_\pi(s)$ is defined by:

$$V_\pi(s) = \mathbb{E}_\pi\big[G_t\big|s\big] = \mathbb{E}_\pi\Big[\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}\Big|s\Big], \tag{2.3}$$

where t is any time step and $\mathbb{E}_\pi[\cdot]$ is the expected value of a variable under policy π. The value of a terminal state is always zero, because there is no expected return after that.

The *action-value function* for policy π is similar to the value function. The only difference is that it is now a function from a state-action pair to a *action-value*. The action-values are also often called *Q-values*. The action-value function specifies how good it is for the agent to take a specific action from a certain state. The Q-value of selecting action a in state s under policy π, denoted by $Q_\pi(s,a)$, is the expected return when the start state is s, the agent selects action a and the agent uses policy π to select actions after that. Then $Q_\pi(s,a)$ is defined by:

$$Q_\pi(s,a) = \mathbb{E}_\pi\big[G_t\big|s,a\big] = \mathbb{E}_\pi\Big[\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}\Big|s,a\Big], \tag{2.4}$$

where t is any time step.

In Reinforcement Learning, the goal is to try to find an optimal policy π^*. A policy π is better than another policy π', that is $\pi \geq \pi'$, if the action-value function of the one is better than the other for all state-action pairs, thus $Q_\pi(s,a) \geq Q_{\pi'}(s,a)$ for all state-action pairs (s,a). If the action-value function is optimal, then the policy that was used must have been optimal. It is possible for two optimal policies to have the same optimal action-value function. The optimal action-value function for state s and selected action a, denoted by $Q^*(s,a)$, is defined by:

$$Q^*(s,a) = \max_\pi Q_\pi(s,a). \tag{2.5}$$

2.3 Learning algorithms

To use the Q-values in the learning process, most of the times, they are stored in a Q-table that keeps track of the Q-value for each possible state-action pair. The number of rows is therefore equal to the number of possible states, and the number of columns is equal to the number of possible actions per state. Of course, there are also other ways to store the Q-values, such as a HashMap.

The reason that the Q-values need to be stored is because this way they are easily retrievable and can be updated again. The updating is done by a learning algorithm. After the learning process, the learned Q-values indicate how good each state-value pair is and what is the best action is to select per state.

In this section, the two most-used learning algorithms are discussed: *SARSA* and *Q-Learning*.

2.3.1 SARSA

SARSA is an on-policy algorithm and stands for $\boldsymbol{s}_t,\ \boldsymbol{a}_t,\ \boldsymbol{r}_{t+1},\ \boldsymbol{s}_{t+1},\ \boldsymbol{a}_{t+1}$. On-policy means that the action a_{t+1} is selected by the policy π. The most-used policy in both SARSA and Q-Learning is the ϵ-Greedy policy, and is explained is Section 2.4. SARSA is characterized by the following equation:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha\big[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\big] \tag{2.6}$$

Here, s_t is the state in the current time step t and a_t is the action selected by the policy π. The immediate reward that follows is r_{t+1}, and the environments transitions into the new state s_{t+1} after executing the action. At last, the policy π is used to determine action a_{t+1} from the new state. The expected return G_t gets approximated by the part

$$r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}),$$

where γ is the discount factor.
The parameter $\alpha \in (0, 1]$ from equation (2.6) is the *learning rate* and determines to what extend new information overrides old information.

2.3.2 Q-Learning

Q-Learning is an off-policy algorithm. This means that the policy π is not used for updating the Q-values, and thus, the policy is only used for selecting the action from the current state. Q-Learning is characterized by the following equation:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha\big[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\big] \tag{2.7}$$

The equation of the Q-Learning algorithm is very similar to the one of the SARSA algorithm. However, the difference is that instead of using the Q-value $Q(s_{t+1}, a_{t+1})$, now the maximum Q-value over all possible actions at state s_{t+1} is used. This way, the action-value function Q directly approximates the optimal function Q^* (Taylor, 2004). The rest of the parameters are the same as described in Section 2.3.1.

The pseudo-code of the Q-Learning algorithm is given on the next page.

Algorithm 1: Q-Learning

Initialize Q-values $Q(s, a)$ for all state-action pairs and set $Q(s, a) \leftarrow 0$
for *each episode* **do**
  Initialize s_t
  for *each time step in episode* **do**
    if s_t *is not a terminal state* **then**
      $a_t \leftarrow$ action given by policy π for s_t
      Take action a_t, observe r_{t+1} and s_{t+1}
      $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\big[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\big]$
      $s_t \leftarrow s_{t+1}$
    else
      break

2.4 Exploration vs. exploitation

Section 2.2.3 introduced policies and Section 2.3 section discussed on-policy and off-policy learning algorithms. Yet, there is nothing said about which policy is actually used in the learning process of the agent. The simplest way of action selection is called Greedy. The *Greedy policy*, as the name suggest, selects an action in a greedy way: the one with the highest Q-value. We can write the Greedy policy as:

$$a_t = \arg \max_{a \in A(s)} Q(s, a), \tag{2.8}$$

where $\arg\max_a$ denotes the action a for which the action value $Q(s, a)$ is maximized. When there are multiple actions with the highest Q-value, then one of them is chosen randomly. The Greedy action selection exploits the available information and knowledge, and it is possible that the agent ends up in a non-optimal policy. This is because it might happen that earlier selected actions are always chosen over actions that lead to undiscovered states and that are actually better in long-term.

There needs to be a balance between exploration of new actions and exploitation of actions that turned out to work good. An often-used policy that handles this problem is called ϵ-Greedy. The *ϵ-Greedy policy* selects actions according to the Greedy policy with probability $1 - \epsilon$. With small probability ϵ, it takes a random action. This way the agent explores new actions. The value of ϵ lies between 0 and 1, and is the trade-off between exploration and exploitation.

Choosing ϵ right makes the learning algorithm converge to the optimal policy π^*. In the first episodes, the agent should explore new actions. As soon as the agent has more knowledge about the environment, we let ϵ gradually reduce, so the knowledge is used for selecting actions.

2.5 Example

In this section, we discuss an example of Reinforcement Learning called the "Cliff Walking" from the book *Reinforcement Learning: An Introduction* by Sutton and Barto, 2018. It highlights the difference between the learning algorithms SARSA and Q-Learning.

This example considers the gridworld environment shown in Figure 2.4. The agent needs to go from the start state S to the goal state G, in such a way that the total reward is as high as

possible. The actions that the agent can take are up, down, left and right. For every action, the agent gets a reward of -1. If the agent falls off the cliff, it gets a reward of -100, and the agent is relocated to the start state S. Since the agent needs to maximize the total reward, it has the urge to walk right next to the cliff, since that is the route leading to the smallest negative reward. However, in this case, the chance of falling off the cliff is high, and thus the chance of receiving a reward of -100 is also high. This urges the agent to not walk right next to the cliff, but this might not always give the optimal solution in terms of maximizing the reward. What the agent will do, all depends on the policy and learning algorithm. The blue safer path will eventually be chosen when using SARSA, the red optimal path continues to be chosen when using Q-Learning.

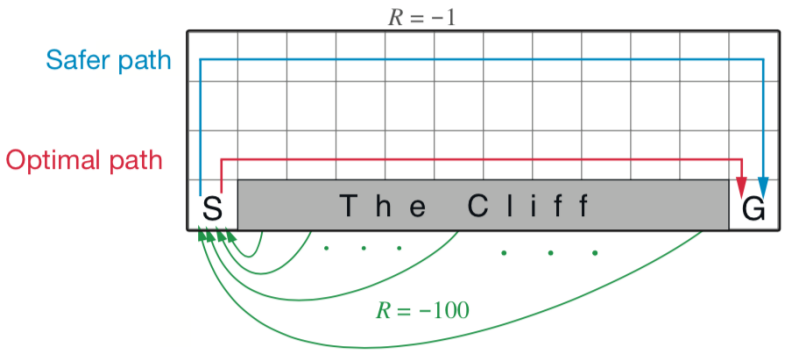

Figure 2.4: The environment of the Cliff Walking example (Sutton & Barto, 2018).

In this example, the policy that is used is the ϵ-Greedy policy, with $\epsilon = 0.1$. Thus, there is a 10% chance that a random action gets selected. The graph shown in Figure 2.5 gives the performances of the SARSA and Q-Learning algorithms. In the first episodes, the performance is pretty much the same. However, the Q-Learning algorithm learns values for the optimal policy, which leads to falling off the cliff. This is visible in the graph, as the total reward is much lower for Q-Learning than SARSA. This is because SARSA learns the agent after enough episodes to take the longer but safer path. Of course, both algorithms would converge to the optimal policy if ϵ would gradually reduce. For example, the Q-Learning algorithm would also stop falling of the cliff if less random actions would be taken, which leads to a higher total reward.

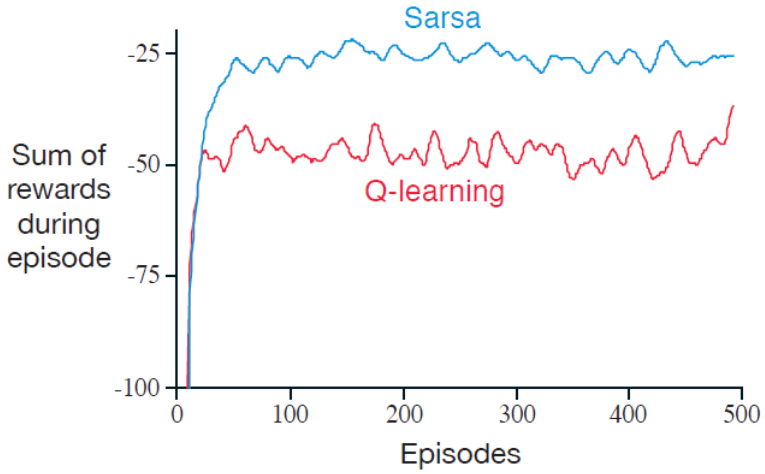

Figure 2.5: A graph of the performance of the SARSA and Q-Learning algorithms in the Cliff Walking example. The total reward per episode is plotted against the episodes (Sutton & Barto, 2018).

Chapter 3

Project

This project focuses on solving a Taxi Dispatch Problem. In a *Taxi Dispatch Problem*, taxis are assigned to trip requests of passengers who are waiting at different locations. In this project, the goal is to solve this problem such that the total waiting time of the passengers is minimized. In this chapter, we discuss how this can be done with Q-Learning. Section 3.1 gives a description of the project's problem in terms of the environment and agent. Section 3.2 discusses the restrictions and assumptions that are made in this project. Section 3.3 gives a formulation of the problem in terms of the elements of a Reinforcement Learning problem as defined in Chapter 2.

3.1 Problem description

The environment consists of an undirected connected graph $G(V, E)$ with V the sets of nodes and E the set of edges. The nodes represent the locations. The number of locations is given by $n = |V|$. The distance between two nodes x and y is equal to the travel time between location $x \in V$ and $y \in V$ and is denoted by t_{xy}. The travel time is symmetric, so $t_{xy} = t_{yx}$. For the travel time from a location to itself, it holds that $t_{xx} = 0$. Furthermore, the number of available taxis m is given and the start locations of the taxis, which are the locations where the taxis are located at the beginning of every episode of the learning process. At last, p denotes the number of passengers that are present at the beginning of every episode, and the origin and destination of these passengers are also given.

In this project, we have only one agent who is responsible for assigning taxis to passengers. Therefore, this agent needs to learn how to make this assignment such that the total waiting time over all passengers is minimized. Every passenger needs to get assigned to a taxi, and it does not matter which taxi. One could say that we look at the problem from the passenger's perspective. We care about how long the passenger has to wait, and thus, we care about how long it takes for the taxi to reach the passenger. Therefore, only the location of the taxi matters, since the travel time between that location and the location of the passenger is given. For example, if the passenger is coupled to a taxi that is at a location five minutes from the passenger, the passenger has to wait five minutes before it is picked up.

3.2 Restrictions and assumptions

Since there exist multiple extensions for the problem, we need to make some assumptions and restrictions. The following assumptions apply to every extension that is made during this project:

(1) The travel time between the locations is given. Since the rewards depend on the travel time from the taxi location to the passenger, we need to know how long this takes.

(2) The location and the destination of a passenger are known. This is because this information is used for learning.

(3) When a taxi is assigned to a passenger, it is unavailable until the passenger is dropped off. This means that the taxi cannot be assigned to any other passenger while it is driving to its assigned passenger, even when the other passenger is at that moment closer.

(4) The destination of a passenger differs from its starting location. This way the taxi always has to go to a different location than where the passenger is picked up.

(5) A passenger can only place a request when a taxi is available. This means that requests coming in when all taxis are busy are lost.

For the initial version, there are two additional restrictions:

(6a) There are no more than p passengers at the same time in the system, where p denotes the number of passengers that is present at the beginning of every episode. This includes the passengers that are assigned to and served by a taxi.

(7a) When a taxi drops off a passenger, and thus becomes available again, a new passenger directly appears in the system.

(8a) When there is a new passenger, it is always assigned to a taxi in the next time step.

Replacing restriction (6a) and (7a) by the following assumptions creates an extension on the previous version:

(6b) There can be more than p passengers, but restriction (5) ensures that there will be no more passengers than taxis.

(7b) From the moment a taxi becomes available, a new passenger can be added at a random time step. Remember that restriction (5) still holds, so a new passenger can appear as long as there is a taxi available.

Another extension can be made to the extension above by replacing restriction (8a) by the following assumption:

(8b) A taxi can choose to not serve a passenger instead of directly serving it.

3.3 Problem formulation

An action is an assignment of passengers to taxis. With taxi, we mean the location of the taxi, since, as stated before, only the location of the taxi matters. If there are for example two taxis with taxi locations T_1 and T_2 and two passenger locations P_1 and P_2, then one of the actions is $\{(P_1, T_1), (P_2, T_2)\}$, which means that passenger P_1 is assigned to the taxi location T_1 and passenger P_2 is assigned to the taxi location T_2. The other possible action is $\{(P_1, T_2), (P_2, T_1)\}$. An extension would be to let the taxi be able to choose not to serve a passenger, since there might appear a new passenger closer to the taxi's current location, and so it is better in terms of total waiting time to let another taxi help the first passenger.

A state is the current state of the environment. It consists of the number of taxis per location and the passengers that are present at that moment. The number of taxis per location are stored in an array of size n, with n the number of locations. The value of the entry is the number of taxis that are available at that location. For example, when there is one taxi at location 0 and one taxi at location 2, then the array looks like: $[1, 0, 1]$.
Passengers are represented by their origin and destination, The origins of the passengers are needed to decide what the possible actions are for that state. To determine what location the taxi ends up in after dropping off the passenger, the state is also dependent on the destination of the passenger. We assume that we always know where the passenger is and where it wants to go. A passenger i, with $i \in \{1, \ldots, p\}$, can be denoted by (o_i, d_i), where o_i is the passenger's origin and d_i its destination, with $o_i \neq d_i$. The value of the origin and destination is the ID of the location. For example, when a passenger is at location 2 and wants to go to location 1, it looks like $(2, 1)$.
Now, consider a state corresponding to the examples above. So, there is one taxi at location 0, one taxi at location 2, and there is a passenger that wants from location 2 to location 1. Then, the state is represented by: $\big([1, 0, 1]; (2, 1)\big)$.

The rewards are the negative waiting times of the passengers. The reward is given when a passenger is assigned to a taxi. For example, suppose that there are two passengers assigned to two different taxis: one with a waiting time of five time steps and one with a waiting time of 3 time steps. Then, the reward is $-(5 + 3) = -8$. If a passenger is not served, it is counted as one time step waiting time, and thus, it gives a reward of -1.

The agent executes the selected action and creates the new state. Now the passengers are assigned to the taxis as stated by the action. The taxis are unavailable while they are driving to a passenger and while they are driving a passenger to its destination. To keep track of this, we create a *job* for every assignment between passenger and taxi. A job keeps track of how many time steps a taxi is already unavailable and what the destination is. Every time step, the job is updated. When a new job is created, the number of taxis at the taxi location is reduced by 1 and the passenger gets removed from the passenger list of the state. Furthermore, the reward is calculated as explained as above. From the already existing jobs, when a taxi reached the destination of the passenger, the taxi is added to that location in the new state. The agent keeps doing this until the episode is ended. Then, the agent is reset. All remaining jobs are deleted. In the new episode, the state of the agent is again the same as the start state.

The learning algorithm that is used is Q-Learning, because the Q-Learning algorithm learns the values of the optimal policy. In contrast to the example in Section 2.5, there is no need to take safer paths in this Taxi Dispatch Problem. Since all the future rewards are considered of same importance, the discount factor is $\gamma = 1$. Furthermore, the learning rate is $\alpha = 0.7$.
The policy that is used to select actions is ϵ-Greedy.

Chapter 4

Results

In this chapter, we consider each of the different versions that are given by the restrictions in Chapter 3 and discuss the results we obtain from our experiment. Section 4.1 gives the experiment setup and defines the convergence of the learning algorithm by two different outputs. In Section 4.2, we analyze these results for each different version of the program and discuss the results and optimal policies.

4.1 Experiment setup

As stated in Section 3.3, the Q-Learning algorithm is used to solve the project's problem. This is, together with the restrictions and assumptions of Section 3.2, implemented in Java. The Java program is coded and run on a laptop with Intel Core i7 6700HQ 2.60 gigahertz with 8 gigabytes RAM.

In the program, we use the policy ϵ-Greedy, which will most of the time select actions based on the Q-values. It chooses the action with the highest Q-value, which means this is the best action to take in that certain state. Of course, this can change during the time the program is running as the agent is gaining new information from the environment. However, at a certain time, the learning algorithm has learned enough and the policy will keep choosing the same actions over and over again. At this point, we say that the learning algorithm has converged to the optimal policy.

To check whether the learning algorithm converges, we run the program ten times and compare the best action per state. By best action, we mean the action with the highest Q-value. If the best actions of the different runs are the same, this means that the learning algorithm has converged. We define convergence by two different outputs:

A. The percentage of states for which the best actions are the same over the 10 runs.

B. The average percentage of runs for which the best action is the same over the states.

The first result, which is *output A*, can be expressed by the following equation:

$$\textit{output } A = \frac{\text{number of states for which the best actions over 10 runs are equal}}{\text{total number of states}} \cdot 100\%$$

The equation that expresses the second result, which is *output B*, is the following:

$$\textit{output } B = \frac{\sum_s \text{ percentage of runs for which the best actions of state } s \text{ are equal}}{\text{total number of states}}$$

If the results are both equal to 100%, the learning algorithm has converged for all states. To get a better understanding, we consider a case in which the learning algorithm is not done learning. Assume that there are 10 states, and for one of them, the best actions are not the same over all the runs, say for only nine of the ten runs. Then, we have $\textit{output } A = \frac{9}{10} \cdot 100\% = 90\%$ and $\textit{output } B = \dfrac{90\% + 9 \cdot 100\%}{10} = 99\%$. With *output A*, it is easy to see that the program should have run a little longer to make the learning algorithm fully converge. However, *output B* shows us that learning process went quite well, as the learning algorithm must have converged for most of the runs, and even did well in the run where it did not converge.

In Section 3.2, multiple extensions are given. We consider each of the different versions and discuss the results. For every version, we can play with the number of locations n, taxis m and passengers p, the number of episodes e and time steps T per episode and the value of ϵ, which results in different values for the outputs.

4.2 Different versions

In this section, we consider all the different versions of the program, state what the impact is on the number of states and state-action pairs, and analyze the results. Each version is an extension of the preceding version, where version 1 is the initial version.

There are a lot of different situations when considering the parameters listed above, at the end of the previous section. Therefore, we work with the simplest situation that still gives taxis the option to move between more than two locations: we begin with three locations, one passenger and two taxis, thus, $n = 3$, $p = 1$ and $m = 2$. As stated in assumption (1) of Section 3.2, the travel time between the locations also needs to be given. The figure below shows the graphical representation of the environment that we work with, where the numbers in the nodes are the index of the locations and the length of the edges is the travel time between the locations.

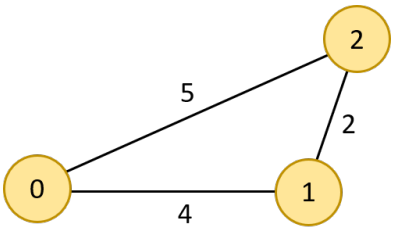

Figure 4.1: A graphical representation of the experiment environment with the location index in the nodes and the travel times between the locations on the edges.

4.2.1 Version 1

The first version, which is the version that forms the base for the extensions, is characterized by restriction (6a), (7a) and (8a) from Section 3.2. So, when a taxi drops off a passenger, a new passenger directly appears in the environment, which again is directly served by a taxi in the next time step. By restriction (6a), this means that there is always one or zero passengers present in the environment. This gives us a total of 39 states. Namely: there are 6 ways to divide the 2 taxis over the 3 locations, where it is allowed to have 2 taxis on the same location. When two taxis are available, there can only be 1 passenger request because of the restriction. Since there are 6 different possibilities for a passenger request, this gives us 6 different states. When

one taxi is available, this means that one taxi is already serving a passenger, and therefore, it is not possible to have another passenger request by restriction (6a). There are three options for the location of the taxi, and there is no new passenger in each of the cases, which gives us 3 different states. As the one taxi left is not able to serve a new passenger while the other taxi is assigned, the case that all taxis are occupied will never occur. This gives us a total of $6 \cdot 6 + 3 \cdot 1 = 39$ states.
With the episodes and time steps per episode set to $e = 150$ and $T = 2000$, and $\epsilon = 0.1$, we get the following result: *output* $A = 100\%$ and *output* $B = 100\%$. So the learning algorithm converges.

Nevertheless, this version of the program is not so interesting. As a new passenger always directly appears when a taxi becomes available, and no other passenger appears since there is a limit, there is nothing unpredictable. The best action is always to serve the passenger and the immediate reward is optimized. In other words, the optimal policy always selects the action that gives the smallest immediate reward, which is equal to the smallest travel time between taxi and passenger.

4.2.2 Version 2

The second version given in Section 3.2 is more unpredictable, as it is characterized by restrictions (6b) and (7b). So, when a taxi drops off a passenger, a new passenger does not necessarily appear directly in the environment. New passengers *are* however still directly served by a taxi in the next time step. Also, as long as there is a taxi available, there is a chance that a passenger request appears. Just as in the previous version, we still have $p = 1$, since there is still only one passenger in the environment at the beginning of each episode. The difference is that there is no limit during the episode anymore. This is because we replaced restriction (6a) by (6b). When there are m_{avail} taxis available, then there are $m_{avail} + 1$ different possibilities for the number of passengers that appear. And each can happen with a chance of $\frac{1}{m_{avail}+1}$. For example, if there are two taxis available, then there either appear two passengers, one passenger or no passengers, each with a chance of $\frac{1}{3}$.
The setting for episodes and time steps of the first version gives *output* $A = 95.26\%$ and *output* $B = 97.79\%$, and thus, the learning algorithm has not converged. Changing the settings to $e = 300$ and $T = 4000$, does make the algorithm converge, as the outputs are both 100%.

The reason that the algorithm does not converge in the same number of episodes and time steps as in the previous version, is because the number of states is now higher. Therefore, the states are not visited as many times as needed to obtain enough information about the environment.
The number of states in this version is 190. When two taxis are available, there can be 2 passenger requests. There are $\sum_{i=1}^{6} i = 21$ combinations of passengers, as there are 6 different possibilities for a passenger request. Of course, it is also possible to have only one passenger request, which gives 6 possibilities, or no passengers, which is also a possibility. When there is only one available taxi, there is also either one or no passengers, and thus, $6 + 1$ possibilities of passengers. When there is no available taxi, the only option is having no passenger requests. The total number of states is therefore: $6 \cdot (21 + 6 + 1) + 3 \cdot (6 + 1) + 1 = 190$ states.

Since the passengers are directly served in the next time step, when there is only one taxi and one passenger, there is only one possible action, which is therefore of course the best action. This is the same for every resulting policy. Also when there are two taxis at the same location, the one passenger is assigned to a location of one of those taxis, which is in either cases the same. That is, there is only one possible action. This is the same for when the two taxis are not at the same

location, but two passengers have the same origin. When the passengers' origins are different and the two taxis are at different locations, the optimal policy mostly selects the action that gives the minimum total waiting time. There are sometimes some exceptions, but there is no clear reason why this is the case for these states. An example is the state $\big([1,1,0];(1,2;\ 2,1)\big)$. The action with the highest Q-value is assigning the taxi at location 0 to the first passenger, whose origin is location 1, and the taxi at location 1 to the second passenger, whose origin is 2. This gives a total waiting time of $4+2=6$, while the waiting time could also be $0+5=5$. When there is only one passenger, it is assigned to the taxi that is closest, and thus, gives the smallest waiting time.

We see that, even though it is now arbitrary when the passengers appear, the optimal policy most of the time selects the actions that result in minimum (total) waiting time. It does sometimes choose the assignment that does not immediately give the smallest waiting time, but not often. It might be because the origin (and destination) of a new passenger are completely random, and so, the chance that a new passenger appears at a certain location is for every location the same. Besides that, the moment that the passenger appears is also random, so it does not become clear for the agent when and where to expect new passengers. This might change when some locations are set as more popular for new passengers to appear. This could be a nice extension. However, with version 3, we first make the picking up process more realistic.

4.2.3 Version 3

In reality, taxis might want to wait a while before serving a passenger, as there is a chance that another passenger request appears and causes a better solution than when the first passenger was directly served. This brings us to a third and last version of the program in this thesis, which is characterized by restriction (8b) of Section 3.2. So, it is exactly the same as the previous version, with the addition that a taxi can 'choose' to not serve a passenger instead of directly serving it.
Again, using the setting for episodes and time steps of the previous version gives $\textit{output } A = 90.53\%$ and $\textit{output } B = 97.74\%$, and thus, the learning algorithm has not converged. This time, it is not because of the number of states, as there are still 190 states, but probably because of the number of state-action pairs, which is increased from 352 to 1666 pairs. Again, there is more time needed to select the possible actions of each state multiple times before the agent has enough knowledge about the environment.
Increasing the episodes and time steps also increases the outputs a little bit. For example when $e = 400$ and $T = 10000$, one of the highest results that are found is $\textit{output } A = 92.11\%$ and $\textit{output } B = 98.26\%$. Unfortunately, it often happens that the outputs are some closer to 90%. Of course there are a lot of different combinations possible, but even the setting of $e = 1500$ and $T = 100,000$ does not give outputs much closer to 100%. Reducing ϵ over time might help, as the agent will make more use of its knowledge. For example, starting with $\epsilon = 0.1$ from the beginning of the run, then reduce it to $\epsilon = 0.05$ from $\textit{episode} = 175$ and further, and at last $\epsilon = 0.025$ from $\textit{episode} = 275$ till 400. One of highest results found is $\textit{output } A = 93.68\%$ and $\textit{output } B = 98.84\%$, which is a little higher, but again, another 10 runs can give lower values for the output. Even though there is a small increment in the highest results found, the outputs are still not equal to 100%. We *can* conclude from *output B* that the percentage of runs for which the best actions are the same for most states are 100%, but from *output A* we know that this is not the case for approximately 8% of the 190 states. It might be that there is a combination of episodes, time steps and ϵ that results in 100% for both outputs, but so far it is not found. The number of episodes and time steps is then probably really high, which makes that the program has to run for a really long time, but so far there is no clear difference between high and low values.

Of course, we can still make conclusions about what the optimal policy is, since *output A* tells us that for approximately 92% of the states the best actions are the same for all runs. When there is only one available taxi, the best action is to help the present passenger instead of waiting. When there are two available taxis and two passengers, only one taxi is assigned to a passenger, and the other passenger is not served. The assignment that is made, is the one that gives the smallest waiting time. However, when there are two taxis and only one passenger, the passenger is still assigned to a taxi, but mostly the one that is the furthest away (except for when the taxi is on the passenger's origin). The states for which the percentage of runs is not 100% are always of the category of two taxis and one passenger.

We see that Q-Learning does work for the first two versions of the program, in which the future is still kind of predictable. However, when the choices really depend on what might happen in the future and the agent tries to learn what is most likely to happen, the learning algorithm does not converge fully. This is maybe because the future is completely random as all probabilities are equal, but there still seems to be a pattern in the optimal policy. As said before, the program might have to run for a really long time to get 100% for both *output A* and *output B*. Therefore, Q-Learning is maybe not the best method to use.

This last version is based on reality the most, and therefore, we would like to continue working with this version of the program or make more extensions on it. Unfortunately, there was not enough time left for this project. An interesting extension would be to let passengers appear even when there are no taxis available, since this makes the program even closer to reality. We could also look at more situations for the number of locations, taxis and passengers for these versions, as there are a lot more locations, taxis and passengers in real life. However, the number of states and state-action pairs would only increase and make it even harder to fully converge, but with the right methods, it would be an interesting extension to look at.

Chapter 5

Conclusion

This thesis focuses on solving a Taxi Dispatch Problem, in which taxis are assigned to trip requests of passengers who are waiting at different locations. In this project, the goal is to solve this problem such that the total waiting time of the passengers is minimized. We tried this with Q-Learning. Together with some restrictions and assumptions, this is implemented in Java, and is used to make an agent learn and generate results.

Since the chosen policy mostly selects the actions with the highest Q-values, also called the best actions, we say that the learning algorithm has converged to the optimal policy as soon as it keeps choosing these best actions. To check this, we run the program ten times and compared the best actions per state. If the best actions of the different runs are the same, this means that the learning algorithm has converged. We define convergence by two different outputs:

A. The percentage of states for which the best actions are the same over the 10 runs.

B. The average percentage of runs for which the best action is the same over the states.

If the results are both 100%, the learning algorithm has converged for all states. The equations that can be used to express these outputs, can be found on page 17. With *output A* it is easy to see whether or not the algorithm has converged for all states. If not, *Output B* shows us how badly the algorithm did not converge.

As the number of states and state-action pairs increase rapidly when increasing the number of locations, taxis and passengers, we looked at the simplest nontrivial situation: three locations, one passenger and two taxis. For this situation, we worked though the three different versions of the program. The initial version is characterized by restriction (6a), (7a) and (8a) of Section 3.2: when a taxi drops off a passenger, a new passenger directly appears in the environment, which again is directly served by a taxi in the next time step. Also, there will always be only one or no passengers waiting. The second version is characterized by restrictions (6b) and (7b): when a taxi drops off a passenger, not necessarily directly a new passenger appears in the environment. New passengers *are* however still directly served by a taxi in the next time step. Also, as long as there is a taxi available, there is a chance that another passenger request appears. The last version is an extension of the second version, which makes the program more realistic: a taxi can 'choose' to not serve a passenger instead of directly serving it.

For each of the versions, the number of episodes and time steps per episode are chosen differently, as later versions have more states and/or state-actions pairs. Therefore, the agent needs more time to visit all the states enough times to gain the right information about the environment and actions.

The outputs are shown in the table below.

	Version 1	Version 2	Version 3
Output A	100%	100%	93.68%
Output B	100%	100%	98.84%

Table 5.1: Results.

For the first two versions, both output A and B are 100%, which means that the learning algorithm has fully converged. In the first version, because passengers are directly served in the next time step, the optimal policy always selects the action that gives the smallest waiting time. The optimal policy of the second method also almost always select the actions that give the smallest (total) waiting time. There are some exceptions, but there is no clear reason why this happens for certain states. The optimal policy therefore seems a little random, but this might be because the appearance of the passengers is completely random. For the third version, there is no combination of values for the number of episodes, time steps per episode and ϵ found that made the algorithm fully converge. It might be that there is a combination that results in 100% for both outputs, but these are probably really high, which makes that the program has to run for a really long time. However, so far there is no clear difference between high and low values, as the outputs are not constant and do not really increase when the number of episodes and/or time steps increase. By *Output B*, we know that for approximately 92% of the states the percentage of runs for which the best actions are the same must be 100%, so we can still make conclusions about the optimal policy. The most interesting occurrences are for the situations with two available taxis present. When there are two taxis and two passengers, only one passenger gets assigned to the closest taxi, and the other one is not served at all. However, when there are two taxis and only one passenger, the passenger is still assigned to a taxi, but mostly the one that is the furthest away (except for when the taxi is on the passenger's origin).

We see that Q-Learning does work for the first two versions. However, when the agent tries to learn what is most likely to happen in the future, the learning algorithm does not converge fully. This also might be because the future is completely random, as all the probabilities in the environment are equal. It would be interesting to extend the program even more to make it more like reality, by for example work with higher numbers of locations and taxis. However, the number of states and state-action pairs would only increase and make it even harder to fully converge. Therefore, Q-Learning might be not the best method to use and it is maybe better to use a different method.

Chapter 6

Discussion

In this thesis, the learning algorithm Q-Learning is used. This is a model-free algorithm, but it was probably better to use a model-based algorithm for the problem of the project.

A *model-based* algorithm constructs a model while learning. A *model* is an element of model-based Reinforcement Learning that simulates the environment and makes predictions about the dynamics of the environment. This way, it is possible to try out possible future sequences of actions without actually performing them in the real environment and see what the outcome would be. Model-based algorithms are mostly used for a planning or prediction problem (integrate.ai, 2018).

A *model-free* algorithm does not use a model and emphasizes learning rather than planning (integrate.ai, 2018). The algorithms are explicitly trial-and-error learners (Sutton & Barto, 2018), and solve control problems (Taylor, 2004). A common control problem is the one with a grid-world in which the agent learns to find its way through a maze. The agent learns what the best action is for each state to reach its goal by trying lots of times.

The Taxi Dispatch Problem in this thesis is much more a prediction problem. The environment is dynamic as the passengers origin and destination are chosen random, and in an extension, even when the passengers appear is arbitrary. By constructing a model, different situations could be simulated and choices could be made on what most likely would happen. Even though trial-and-error works too, the approach of model-based Reinforcement Learning would make more sense.

We can also see this in the results. In the last version of the program, the taxis have the option to not serve a passenger, with the idea that waiting a little longer might give a better assignment and would not increase the waiting time too much. We saw that the learning algorithm got quite close, as both outputs A and B were always between 90 and 100%, but yet, it did not succeed in fully converging so far. It might be that it does when the number of episodes and time steps are set really high, but that would make the program to have to run a long time. Therefore, as the agent tries to learn what is most likely to happen, a model-based method might help to speedup the learning process.

Bibliography

[1] Alshamsi, A., Abdallah, S., & Rahwan, I. (2009). Multiagent Self-organization for a Taxi Dispatch System. In Decker, Sichman, Sierra & Castelfranchi (Eds.), *Proc. of 8th Int. Conf. on Autonomous Agents and Multiagent Systems (AA-MAS 2009)* (pp. 21-28). Budapest, Hungary.

[2] Buşoniu, L., Babuška, R., & De Schutter, B. (2010). Multi-agent reinforcement learning: An overview. Chapter 7 in D. Srinivasan & L.C. Jain (Eds.), *Innovations in Multi-Agent Systems and Applications - 1* (pp. 183-221). *Studies in Computational Intelligence* (Vol. 310). Berlin, Germany: Springer.

[3] Crites, R. H., & Barto, A. G. (1998). Elevator Group Control Using Multiple Reinforcement Learning Agents. In M. Huhns & G. Weiss (Eds.), *Machine Learning (1998)* (Vol. 33). (pp. 235-262). Boston, MA: Kluwer Academic Publishers.

[4] Dietterich, T. G. (1999). Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition.

[5] Güldenring, R. (2019). *Applying Deep Reinforcement Learning in the Navigation of Mobile Robots in Static and Dynamic Environments* (Master's thesis). Universität Hamburg, Hamburg, Germany.

[6] Integrate.ai. (2018). What is Model-Based Reinforcement Learning?. Retrieved from `https://medium.com/the-official-integrate-ai-blog/understanding-reinforcement-learning-93d4e34e5698`

[7] Kuo, M. (2016). Taxi Dispatch Algorithms: Why Route Optimization Reigns. Retrieved from `https://blog.routific.com/taxi-dispatch-algorithms-why-route-optimization-reigns-261cc428699f`

[8] Russell, S. J., & Norvig, P. (2010). *Artificial Intelligence: A Modern Approach* (3rd ed.). Upper Saddle River, NJ, USA: Prentice Hall.

[9] Sharma, A. (2018). *The AI Circle* [Image]. Retrieved from `https://www.datacamp.com/community/tutorials/machine-deep-learning`

[10] Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction* (2nd ed.). Cambridge, MA, USA: MIT Press.

[11] Taylor, G. W. (2004). *Reinforcement Learning for Parameter Control of Image-Based Applications* (Master's thesis). University of Waterloo, Waterloo, Ontario, Canada.

[12] Van Otterlo, M., & Wiering, M. (2012). Reinforcement Learning and Markov Decision Processes. Chapter 1 in M. van Otterlo & M. Wiering (Eds.), *Reinforcement Learning: State-of-the-Art* (pp. 3-42). *Adaptation, Learning, and Optimization* (Vol. 12). Berlin, Germany: Springer.

[13] Verma, T., Varakantham, P., Kraus, S., & Lau, H. C. (2017). Augmenting Decisions of Taxi Drivers through Reinforcement Learning for Improving Revenues. *Proc. of the Twenty-Seventh Int. Conf. on Automated Planning and Scheduling (ICAPS 2017)* (pp. 409-417). Pittsburgh, Pennsylvania, USA: AAAI Press.

[14] Wang, J., & Lampert, B. (2014). Improving Taxi Revenue with Reinforcement Learning.

[15] Zander, G. (2017). *Predicting taxi passenger demand using artificial neural networks* (Master's thesis). KTH Royal Institute of Technology, Stockholm, Sweden.