Computing Science

Choosing taxicab fare prices using a Q-Learner

Karlis Venters

Level 4 Single Honours Project (45 Credits)

Department of Computing Science
University of Aberdeen
King's College
Aberdeen AB24 3UE

May 2014

Declaration

I declare that this document and the accompanying code has been composed by myself, and describes my own work, unless otherwise acknowledged in the text. It has not been accepted in any previous application for a degree. All verbatim extracts have been distinguished by quotation marks, and all sources of information have been specifically acknowledged.

Karlis Venters

Date: May 15, 2014

Abstract

Taxi fare regulation and fixed taxi fare prices are inefficient. This project suggests a variable fare pricing approach to improve taxi profitability, where the prices are chosen using reinforcement learning. Specialised simulation software called Taxi-sim was developed for this project and published online. The user of Taxi-sim can specify environmental constraints and market characteristics, and taxi agents in the simulation are operated by a simple Q-Learner. The experimental simulation results show that the Q-Learner was able to improve its profit over time when using variable fare pricing, and that its profits were significantly larger than when it used fixed fare pricing. However, the analysis of fixed pricing data showed no profitability and no profitability improvement over time, either suggesting that the Q-Learner is not suitable for fixed pricing or revealing an implementation bug. This project concludes that reinforcement learning can successfully be used for setting taxi fare prices and justifies further study in the field. Some of the future work can be done using the Taxi-sim simulation software which has been designed to be modular and extensible, however it is important to be aware of its limitations and known bugs.

Acknowledgements

I would like to thank my supervisor Dr. Nir Oren for all the advice and assistance he provided over the course of the project.

I would also like to thank the people who advised me and helped with proofreading the report.

Contents

List of Figures

1 Introduction

1.1 Overview

This is a report on an experimental simulation and the accompanying software in machine learning and taxi economics. It was written in support of an undergraduate project for BSc Computing Science degree at University of Aberdeen.

Taxis (in some literature the term *taxicab* is used) are an important part of transport systems worldwide and usually are regulated by local governments. However, fare regulation creates economic inefficiencies because the demand and supply greatly differs based on many different factors. Therefore no regulation could be preferable when the taxi's fare price is set freely. This project introduces Taxi-sim, a software for taxi market simulation where Q-Learning is used to set taxi's fare prices.

The report starts by setting main and optional goals for the project in Section 1.3. Detailed background information on the problem is given in Section 2, investigating transportation economics, reinforcement learning and related works. The core requirements for a software implementation are established in Section 3. Section 4 describes the chosen software development approach as well as the architecture and technologies that are used. The finished software is discussed in sections on implementation (Section 5.1) and testing (5.2). The simulation experiment with its results and the overall success of the project are discussed in Section 6. Finally, the project is summed up in Section 7.

1.2 Online

The project has a homepage which also serves as the main web page for the source code repository, located online at this address: `https://github.com/vencha90/taxi-sim`. Please use the project homepage to find the most up-to-date URLs if any of the URLs specified in the report are not working. The results of the experiment are available online at this address `https://db.tt/zYPt09CL`.

1.3 Goals

The project hypothesis is that taxicab profits can be improved by adapting a dynamic pricing strategy, instead of charging fixed fares calculated by a taximeter depending on time and distance. The prices can be set by an AI system using reinforcement learning with increased profit being the ultimate goal.

The suggested approach to investigate this problem is an experimental software simulation. A reinforcement learning controlled taxi agent travels in some environment and serves passengers with the aim of maximising its profit. The environment, passengers and the taxi can be configured as needed for research.

To establish whether the approach worked, it needs to be evaluated. This can be done by comparing the results with a benchmark of running the simulation with an identical environment – this time with fixed fares. Now a clear set of goals can be extracted from the short project vision described above:

- **Background.** Research literature and related works to find a suitable approach to solving the

problem.

- **Software.** Develop a software simulation of a taxi market. The simulation needs to support regulated and unregulated pricing.
- **Data.** Ensure that essential data is gathered and analysed. Compare the performance of agents working under price regulation and no price regulation to determine whether this project's premise is worth any future study.
- **Extensibility.** Develop the software to be modular and maintainable in order to be able to simulate various environmental conditions, agents and reinforcement learning strategies.

Further lower-priority goals that are beneficial to the core goals can be identified as well:

- **Reinforcement Learning Strategy.** Compare various reinforcement learning strategies,
- **User Interface.** Develop an user-friendly interface.

2 Background

Pricing is one of the main problems in transportation due to its large social impact, whether it be road tolls, bus prices or regulated taxi fares. One of the most controversial topics in the taxi market is about whether market regulation is necessary. There is no consensus on this question, and different authors give contradictory opinions based both on theoretical studies and factual real world research. The economics of transportation are covered in Section 2.1.

The next part of background that needs covering is artificial intelligence and reinforcement learning in particular. The various ways of modelling reinforcement learning problems are investigated in Section 2.2.

Finally, related works are discussed in Section 2.3, spanning both artificial intelligence and transportation economics. A recent trend with promising results for transport planning and pricing is the usage of microeconomic models based on individual agents, allowing variable pricing to be used thus improving system efficiencies. Following this trend software has been developed to simulate such agent-based scenarios, the most notable of which is MATSim.

2.1 Transportation Economics

Taxis (also known as *taxicabs*) are an important part of public transportation. Because of their prevalence worldwide and importance in transport systems a wide range of literature has been produced on taxis. Three different types of taxi markets can be distinguished: cruising taxi market when a passenger hails the taxi on street with visual contact, phone-order taxi market, and taxi ranks where multiple taxis wait for passengers. Unless specfically noted, the discussion in the following sections applies to all of them.

A major topic in the research and discussion on taxis is taxi market regulation, and parts of it are relevant and will be considered in some detail in Section 2.1.1.

For this project, the most relevant area of these works is economic modelling of taxi markets, an overview of which is given by Salanova et al. (2011). Discussion on Economic modelling is in Section 2.1.2.

Finally, a look at the relationship between supply and demand (Section 2.1.3 is needed before any actual modelling can be performed.

2.1.1 Regulation

Regulation is a controversial topic in taxi market research as no consensus has been reached on whether it is recommended.

Cairns and Liston-Heyes (1996) investigated economic workings of taxi markets and incorporated results of earlier research in their economic equilibria findings. They concluded that regulation is needed to achieve non- negative profits (the so-called economic second best).

OECD (2007), cited by Salanova et al. (2011) listed arguments both for and against regulation as observed in different countries, and noted that markets with widely varying regulation can operate successfully. It is important to note that some markets considered *deregulated* still have some form of fare regulation, for example, taxis in New Zealand are required to list their maximum fares based on

time and distance, but are not forced to follow them (Gaunt 1995).

2.1.2 Economic modelling

Taxi market modelling was first done by Douglas (1972), according to Salanova et al. (2011). He investigated a regulated cruising-taxi market and defined the fundamental taxi problem to be finding an equilibrium of an optimal level of service matching an optimal price. His limited model has been used as reference by all the later authors cited by Salanova et al. (2011) that have extended it to other taxi markets and factored in more environmental influences.

De Vany (1975) researched regulated taxi markets organised as a franchised monopoly, using a medallion system, and having free entry. With the goal of finding equilibrium output, capacity and utilisation he suggested a formula for passenger demand depending on taxi fare, passenger value of time and waiting time. Manski and Wright (1967) analysed the taxi market from a purely economical point of view and concluded that in addition to exogenous variables, passenger demand for taxi services is also directly related to taxi supply through waiting time. Similarly, taxi supply is influenced by taxi utilisation, which in turn directly depends on passenger demand.

The most recent publications in this area are sophisticated models based on the network model for cruising-taxi market by Yang and S. Wong (1998). This network was modelled as a graph and assumed constant taxi demand and supply, passenger demand was represented as origin-destination matrices. Finally, this paper suggested an algorithm to find an equilibrium for the optimal number of taxis in a market and equations to calculate taxi utilisation and customer waiting time.

In contrast, Yang, Lau, et al. (2000) focused on supply and demand to recommend optimal policies for taxi regulation in Hong Kong and based their model on various data sources. A number of exogenous and endogenous variables affecting taxi market were identified, and equations were suggested to calculate them: passenger waiting time, percentage of occupied taxis, vacant taxi headway, number of daily taxi passenger trips and taxi waiting time. This model can be used to forecast taxi demand, taxi utilization and service quality, although the authors warned that it does not take in account all of the complex supply-demand relationships in taxi markets.

Consequently Yang, S. C. Wong, and K. Wong (2002) continued to evaluate the supply- demand equilibria of taxi market started by Yang and S. Wong (1998) and Yang, Lau, et al. (2000), resulting in the conclusion that the spatial characteristics of a network where taxis are operating strongly influence supply and demand, and should bear weight when evaluating regulatory policies.This study focused on social surplus (the sum of customer surplus and producer surplus) as the key objective of taxi markets. Four different regulatory frameworks that could be applied to taxi markets were investigated: free entry and unconstrained fare, free entry and regulated fare, regulated entry and unconstrained fare, and regulated entry and regulated fare. All of these cases were investigated with both competitive and monopolistic markets, and equilibria were found.

K. Wong et al. (2008) extended this model to heterogeneous vehicle and user classes, and included congestion which is a major issue in reality but was ignored by earlier research. Yang, Fung, et al. (2010) proposed a non-linear fare structure to correct market and regulatory inefficiencies, and

applied it to a similar model.

The way how cruising taxis and customers find each other was researched by Yang, Leung, et al. (2010), paying particular attention to customer behaviour: this study permitted customers to use other modes of transport e.g. public transit or walking to find taxis and/or reach their destinations. However, this study was based on a taxi market with fixed fares. When fares can be negotiated, taxis and customers are likely to do some bargaining over the amount of fare which would affect their transport mode preferences.

Cairns and Liston-Heyes (1996) gives a quick overview of bargaining and its implications in different markets, with customers having more power in taxi rank and phone order markets due to easily available competition, and high costs to search for alternatives in cruising taxi market for both parties resulting in higher willingness to agree. Bargaining of minimal-intelligence agents in competitive markets was investigated and implemented in software by Cli (1997), where a bargaining performance similar to humans was achieved. Rubinstein (1982) described equilibria for a bargaining model where each round of bargaining has costs to participants.

2.1.3 Demand and supply

Taxi demand is a part of the total demand for transportation. There are two approaches to modelling demand for public transportation: aggregate and disaggregate. Aggregate models are macroeconomic, while disaggregate models are microeconomic and based on the individual agents in a system. Recently disaggregate models have emerged as the main method of modelling demand, but these models require detailed microeconomic data for a system. Because of difficult processing of the large datasets, applying disaggregate models usually still involves some form of aggregation. Customers' value of time (VOT) and value of reliability (VOR) are the most important quantities determining the demand for public transportation, the sum of whom are a customer's total willingness to pay for some trip. Both VOT and VOR derive on customers' characteristics and environment they are in, for example, their income, whether the planned trip is for pleasure or a commute to work, and even the tax rates. Aggregate models using VOT and VOR have been developed as well, although VOR has been researched significantly less. (Small and Verhoef 2007)

Yang, S. C. Wong, and K. Wong (2002) cites Manski and Wright (1967) on the complex structure of demand in taxi markets, shown in Figure 1. Both taxi demand and supply are influenced by exogenous variables (and regulation policies, if any). Taxi demand influences taxi availability and vice versa. Similarly, taxi supply influences taxi utilization and vice versa. Taxi demand influences taxi utilization and thus indirectly influences supply, similarly taxi supply influences taxi availability and thus indirectly influences demand.

Customer demand is modelled as a function of waiting time and fare price in many studies: Douglas (1972), De Vany (1975), Cairns and Liston-Heyes (1996), and Yang, S. C. Wong, and K. Wong (2002) all used customer waiting time as a proxy for service quality. According to Salanova et al. (2011), Manski and Wright (1967) used a Poisson process (a stochastic function) to simulate demand.

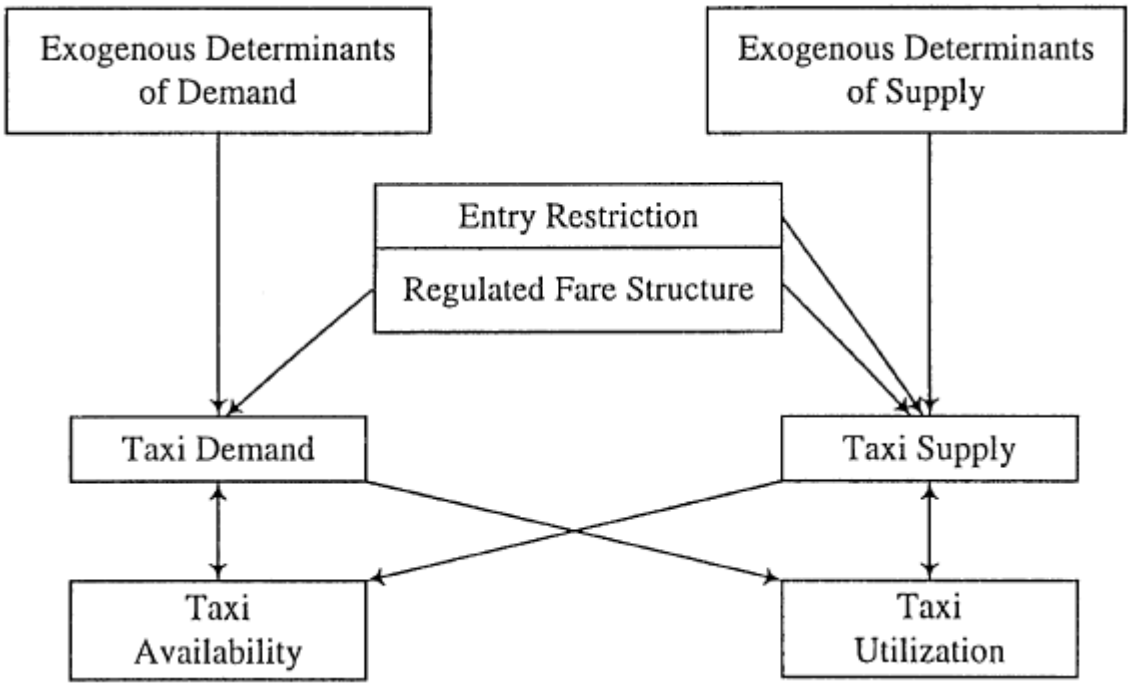

Figure 1: Demand-availability-utilization-supply relation in a taxi market. Manski and Wright (1976), cited by Yang, S. C. Wong, and K. Wong (2002).

Yang, S. C. Wong, and K. Wong (2002) use a disaggregate demand model (separately for each origin-destination pair), where waiting time depends on the number of vacant taxis in an area near the customer and price depends on the distance covered; Yang, Fung, et al. (2010) added travel time as an additional variable indicating service quality and assumed that demand decreases as waiting time increases. Yang, Leung, et al. (2010) took a slightly different approach by modelling customer demand as their willingness to pay to reach a destination, based on their subjective monetary value for using different modes of transport for reaching a destination; therefore the demand for taxis in this study was only a part of the total demand for transportation.

2.2 Reinforcement Learning

2.2.1 Basics

Reinforcement learning is software agents being rewarded or punished (receiving negative reward) for their actions so that they can hopefully learn an optimal policy for acting in some environment, maximising the economic outcome (**utility**).

Optimal policy is defined as the mapping of actions from any possible state to the highest utility outcome from that state. Thus for each state there is a maximum utility that can be reached. Agents have a set of possible actions that they can take, depending on the state they are in. Agents also have a set of rules for what they can observe from the environment. **Transition model** is the description of how applying actions to states result in new states. All of the states, actions and the transition models together define the **state space**. (Russell and Norvig 2010; R. S. Sutton and Barto 1998)

Reward is the relative utility (positive or negative) what an agent gets for being in a certain state, and the goal of the agent is to maximise the cumulative reward over time. The cumulative reward is known as the **value function** or **utility function**, giving a long-term outlook in contrast to immediate rewards. Searching and learning the value function is central to reinforcement learning. (Russell and Norvig 2010; R. S. Sutton and Barto 1998)

Evolutionary methods (R. S. Sutton and Barto 1998) (also referred to as genetic algorithms by Russell and Norvig (2010)) are a branch of artificial intelligence that could potentially be used for solving reinforcement learning problems. Evolutionary methods do not learn from individual experiences, and therefore R. S. Sutton and Barto (1998) do not recommend using them for reinforcement learning problems, although they agree that evolutionary methods could be useful in specific cases. Similarly Russell and Norvig (2010) stated that genetic algorithms do not cope well with increasing complexity and are the most useful in optimisation problems.

2.2.2 Markov Decision Processes (MDPs)

An agent having the **Markov property** means that only the current state of the agent matters to the probability distribution of future states i.e. any other states and actions do not influence the future state. Even in cases when the Markov property does not strictly apply, often approximations can be made and reinforcement learning theory still applies. (R. S. Sutton and Barto 1998)

A process satisfying the Markov property is a **Markov decision process (MDP)**. It consists of the state space and reward functions. It can be seen that all MDPs have a policy, the one with the highest expected utility called the **optimal policy**. It is important to note that it is the *expected* utility because the environment could be (and usually is) stochastic. (Russell and Norvig 2010)

Considering MDPs over time, two cases can be distinguished: finite and infinite horizon. While the optimal action in a given state could change when having a finite horizon, it never would with an infinite horizon as there is no time pressure. To solve MDPs with an infinite horizon and no terminal states, rewards need to be discounted using a discount parameter γ. Finding an optimal policy is not always possible due to the large state space, therefore approximation may be needed. Approximation can give good results because reaching many states has a very small probability and they have very low effect on the overall utility of a policy. (Russell and Norvig 2010)

The simple model of MDPs assumed that the environment was perfectly observable i.e. the agent knew which state it is in at all times. This assumption is unrealistic in the real world – for example, a taxi driver does not know when they will find a passenger and if the passenger will accept their offered fare. In this case the environment is partially observable, hence the problem could be defined as Partially Observable MDP (POMDP). This case is considered in Section 2.2.6

According to Spaan (2012), fully observable finite MDPs can be formally defined as follows. Time has discrete steps, and an agent has to execute an action at each step (unless the action lasts multiple steps). The environment is described by a set of states $S = \{s_1, .., s_n\}$, and the agent has a set of actions $A = \{a_1, .., a_n\}$. Each time the agent takes an action a in a state s, the environment transitions to a new state s' according to a probabilistic transition function $T(s, a, s')$, and the agent

receives an immediate reward $R(s,a,s')$.

It can be concluded that the problem of this paper could be a POMDP and could be the simpler MDP, depending on the specification. There is a clear reward function between states - the fare a passenger pays. The agent (taxi) may or may not be aware of the full model of the environment. If passengers are waiting, taxis not in the immediate vicinity will now be aware of it. Considerations of solving MDPs will be examined in Sections 2.2.3 and 2.2.5 with an in-depth overview of methods in Section 2.2.4. Finally, POMDPs that could be important for the future outlook of this project will be looked at in Section 2.2.6

2.2.3 Solving MDPs

Two groups of approaches for solving reinforcement learning problems (finding optimal policies) were covered by both Russell and Norvig (2010) and R. S. Sutton and Barto (1998) and can be categorised as model-free methods and model-based methods (discussed in Section 2.2.4). However, both types of methods have a common background that first needs to be looked at in this section.

The simplest naive approach to solve reinforcement learning problems is **direct utility estimation**. It uses the fact that utility of a state is the expected total reward from that state onward, called the reward-to-go. (Widrow and Hoff, 1960 cited by Russell and Norvig (2010)) After a large number of estimations giving samples for the states, the observed reward-to-go is likely to converge to the actual utility of a state. However, this approach is inefficient, mainly because it ignores that utilities of states are related to each other. (Russell and Norvig 2010)

Dynamic programming methods, unlike direct utility estimation, take in account the interdependence of utilities of states by learning the transition model that connects them, and uses dynamic programming to solve the corresponding Markov decision process. However, this requires a perfectly known model of the environment which in practice is infeasible. Dynamic programming is also computationally expensive. The simplest dynamic programming method is value iteration. Dynamic programming is also the basis for the advanced methods looked at in Section 2.2.4. (R. S. Sutton and Barto 1998)

Dynamic programming methods are based on generalized policy iteration (GPI) which has been proven to converge to optimal policies and value functions. GPI repeatedly alternates between these two steps: policy evaluation and policy improvement. Policy evaluation calculates a value function based on the current policy and updates the existing value function to be closer to the newly calculated value function. Policy improvement uses the updated value function to calculate a new policy and then updates the existing policy to be closer to the newly calculated policy. Most reinforcement learning problems use GPI. (R. S. Sutton and Barto 1998)

2.2.4 Traditional MDP Methods

Two groups of methods are distinguished by R. S. Sutton and Barto (1998): *off-policy* and *on- policy*. Off-policy methods are called that way because they do not operate based on the estimated optimal policy, but have a behaviour policy that could even be completely unrelated to the estimated optimal policy. This allows for exploration, while an on-policy agent operates on and estimates a single policy.

If an agent does not explore, it can easily settle on a sub-optimal policy.

Temporal difference (TD) methods do not use a transition model and therefore are simpler and require less computation than basic dynamic programming, alas at a price of slower learning. TD works by adjusting the utility estimates based on the differences observed in the last state transition, and over time the *average* utility values converge to the correct values. To ensure that utility values in TD converge to the correct value, visited states can be stored and their repeated impact reduced. (Russell and Norvig 2010)

To take in account more of the observed history than just the last state transition, TD(λ) is named by (Sutton1998ai+reinforcement) as an improved method. The parameter λ regulates how much of the transition history is used when updating policy to correct for errors observed in the last transition. (Szepesvari2010ai+algorithms) notes that finding the best value of λ is usually done by trial and error. However, this only affects the speed at which the method converges to an optimal policy, and λ can be adjusted as needed when some experience of using the method is known.

A popular off-policy model-free method is **Q-learning**, which is based on the TD approach. A Q-learning agent works by learning an action-value function for each state that ultimately gives the expected utility of taking a given action and following the optimal policy thereafter. The optimal policy can be constructed by simply selecting the action with the highest value in each state when the action-value function is learned. Q-learning is proven to converge to optimal policies over time. Unfortunately Q-learning is believed to be limited in complex environments as it cannot simulate possible courses of action. However, Q-learning is probably the easiest reinforcement learning method to implement. (Russell and Norvig 2010)

The space that an agent operates on could be very large, for example, having a near infinite set of states or actions. In the case of having multiple states function approximation is suggested by R. S. Sutton and Barto (1998) and Russell and Norvig (2010). Szepesvri (2010) notes that it is not known if function approximation when used for Q-learning will converge to optimal policies, although it has been proven to work and converge to an optimal policy in very specific restricted conditions. A different approach for continuous state and action spaces using TD actor-critic Monte Carlo methods is described by Lazaric, Restelli, and Bonarini (2008), where the software agent has a separated policy and value function.

2.2.5 Exploration

A final issue not yet covered is exploration. GPI-based methods (see Section 2.2.3) covered here in their naive forms are greedy, meaning that they will often converge to a policy optimal for the subset of states they have experienced (a suboptimal policy) (Russell and Norvig 2010). Instead, the agent should find a globally optimal policy. Thus the agent faces a trade-off between exploration and exploitation i.e. between current profits and future profits.

R. S. Sutton and Barto (1998) describe ε-greedy methods to deal with this trade-off. This simply fixes how inclined an agent is to choose an action which it considers optimal with a variable ε, choosing a random action with a probability $1-\varepsilon$. If the environment does not change over time, then it is

suggested to reduce ε as the agent is more likely to have found a good policy.

However, taking an action randomly after some experience with the environment is naive. A common method for discrete action spaces is Boltzmann exploration. The Boltzmann exploration strategy privileges the execution of actions with higher estimated utility values, increasing the bias towards greater exploitation over time. Gaussian exploration is a similar approach for continuous action spaces. (Wingate 2012)

2.2.6 Partially Observable Markov Decision Processes (POMDPs)

Partially observable Markov decision processes (POMDP) add the notion of a **sensor model** to specify the probabilities of perceiving evidence. Therefore now the set of states is a set of probability distributions for a POMDP – the belief state as discussed by Russell and Norvig (2010). Spaan (2012) go in more detail and specify that to form belief models, an agent needs some form of memory to record observations. With continuous states or with large planning horizons storing the history directly is infeasible.

Russell and Norvig (2010) noted that to use reinforcement learning methods on POMDPs, approximations need to be used for the continuous state space. For the transformation of POMDP to a MDP Spaan (2012) suggested a method developed by Stratonovich (1960), Dynkin (1965) and Astrom (1965): summarising agents' observation history to a belief vector b of continuous probability distributions.

Q-learning and SARSA that were mentioned in 2.2.4 have been adapted to work with POMDPs according to both Russell and Norvig (2010) and Spaan (2012), alas with limited success. These POMDP methods are extensions of the basic MDP methods much like POMDPs are extended MDPs, therefore an evolutionary transformation of this project from using MDP to using POMDP is likely to be possible if ever needed. A detailed overview of these and other POMDP methods is given by Spaan (2012).

2.3 Related works

Although the subject area of the project is rather specific, some related works were found. Research studies are covered in Section 2.3.1, traffic simulation software in Section 2.3.2 and practical guidance on traffic modelling in Section 2.3.3.

2.3.1 Studies

Emele et al. (2013) investigated an agent-based scenario for variable public transport pricing where both passengers and transport providers submit requirements and bids to an intermediary agent. Transport vehicles can be shared by multiple passengers, and passengers have hard and soft requirements, with soft requirements ordered by importance (although only the first soft requirement is taken in account by this system). This paper concluded that variable pricing results in lower fares and more served passengers than a fixed pricing scenario.

Neumann and Balmer (2011) researched how top-down aggregate and activity- based microscopic agent-based models can be used in Berlin. It was found that the both types of models used are complementary and provide similar results most of the time. This study used MATSim, a piece of

software described in detail in Section 2.3.2. Although this study did not cover pricing in any way, it did discover that both bottom-up and top- down approaches provided interchangeable results.

2.3.2 *Traffic Simulation Software*

Historically the most important traffic simulation software products have been TRANSIMS (Smith et al. 1995) and EMME (Gao, Balmer, and Miller 2010). TRANSIMS operates on aggregated data in a top-down fashion, and is limited to certain software modules and specific inputs operating, and does not support agents. TRANSIMS proved to be useful at its time, but agent-based microsimulation software has been more popular recently because of the richer supported functionality. (Bernhardt 2007)

EMME in its latest version 4 is a commercial software, and potentially the most advanced according to the claims of its parent company. However, it is of limited interest for this project as EMME is a commercial and closed software. (INRO 2014)

Bernhardt (2007) gave an overview of agent-based software for traffic modelling, naming MATSim as the most useful of these. MATSim is the most noteworthy agent-based simulation and is undergoing active development. It has very extensive capabilities and modules. However, it is complex to use and extend. Gao, Balmer, and Miller (2010) claimed that MATSim and EMME/2 has similar performance and even produce compatible outputs.

ITSUMO by Silva et al. (2006) is more limited simulation software using intelligent agents to model traffic. Bazzan, Nagel, and Klgl (2009) integrated MATSim and ITSUMO to produce rich large scale traffic simulations. MATSim, ITSUMO and the integrated piece of software are all meant to be extendible. Unfortunately none of them have in-built support for agent interactions that involve prices.

MATSim is arguably the most advanced of these open-source software products. It has a large codebase under active development, and its documentation is fragmented matsim.org (2012). Extending MATSim is conceptually a good alternative to implementing custom software for this project. However, the effort required to fully investigate the complexities of MATSim and work with an unfamiliar legacy codebase was likely to prove infeasible for the project's time frame. MATSim is developed for the purposes of traffic prediction, not reinforcement learning simulations.

2.3.3 *Traffic Modelling*

Bernhardt (2007, p. 78) cited Bonabeau(2002a) for the following conditions when agent based modelling is appropriate:

- When there is potential for emergent phenomena–that is, individual behaviour is nonlinear and changes over time, and fluctuations are more important than averages;
- When describing the system from the perspective of its constituent units activities is more natural–that is, activities are a more natural way of describing the system than processes; and
- When the appropriate level of description or complexity is not known ahead of time.

Finally, if an agent-based approach has been deemed appropriate, modelling can be started. To create models the following needs to be carefully defined: the environment; agents with their types, attributes, allowable and initial values for the attributes; agent interaction rules with each other and the environment. (Bernhardt 2007)

3 Requirements

3.1 Functional Requirements

The following requirements are derived directly from the project's goals set in Section 1.3. Alas, it is infeasible to structure this section around each goal as they contain several requirements that sometimes overlap with other goals. Each of the following requirement categories is discussed in detail, listing what could be implemented in the best case scenario without any time pressure. However, to stay realistic and make sure that core functional requirements are prioritised, the minimal requirements are separately summed up in Table 1 for each of the following sections.

3.1.1 Market Considerations

The approach suggested by this project is not compatible with a market where fares are regulated, at least in the usual form of regulation that specifies a formula to calculate fares based on some variables, usually time and distance. Other ways of regulation that do not affect pricing are compatible with the suggested variable pricing approach, for example, regulated market entry conditions.

Three different operational types of taxi markets were introduced in Section 2.1: phone-order market, cruising taxi market and taxi rank market. In the phone-order market and taxi rank customers are actively seeking a taxi, while in the cruising taxi market passengers can only wait for a taxi to drive by. Therefore the cruising taxi market is chosen as the easiest target for simulation, investigating extensions to phone-order and taxi rank markets if the initial experiment is successful.

The relationship between taxi demand and supply and vice versa was established in Section 2.1.3. This relationship largely depends on the competitive situation in a taxi market. This is a very complex relationship and thus implementing it in software would require considerable effort. Because the core goal of the project is comparing variable pricing approaches, not correctly modelling the market in every aspect, this problem can be avoided by assuming a constant demand. Furthermore, if the environment is fixed, the simulation only needs to support a single taxi at the very least. As the design is planned to be extensible, the variable demand can be factored in later when necessary.

3.1.2 Demand: Modelling Passengers

Customer demand was reviewed in Section 2.1.3 where two approaches to modelling demand were shown: aggregate and disaggregate. An aggregate demand model "for some portion of the travel market is a function of variables that describe the product or its consumers" (Small and Verhoef 2007) The disaggregate approach specifies a set of variables for each individual passenger. The project goal is investigating taxi pricing on an individual basis, therefore an agent-based approach is preferable as it allows individual modelling of passengers. Of course, to express demand as a single variable it needs to be an aggregation of the relevant individual variables.

Therefore each passenger's demand is a function of some variables. The different types of variables affecting taxi demand were discussed in Section 2.1.3. Exogenous demand variables are value of time and value of reliability that can be derived from a passenger's income, hour of day when travelling, purpose of travel, social status, cost of waiting and others; these can be modelled for each individual passenger. Taxi availability is a variable directly depending on the number of vacant taxis

in an area, this is something that passenger's perceive in reality. However, for the simulation, taxi availability needs to be assumed a constant, at least until machine learning capabilities are added to passengers and a competitive market established beyond the very basic simulation, so that passenger agents can learn the availability on their own.

Let P be the set of relevant characteristic variables $c_1,..,c_n$ for a passenger: $P = \{c_1,..,c_n\}$, where each $c_i : i \in \{1,..,n\}$ has a function $f_i(c_i)$ that returns a unity-based normalised value of c_i, and each c_i has a weight w_i representing its relative importance compared to other variables, where $\sum_{i=1}^{n}(w_i) = 1$.
Then the total probabilistic value $V(P)$ of a passenger is

$$V(P) = \sum_{i=1}^{n}(w_i \cdot f_i(c_i)))$$

Let $\Delta(o,d)$ be the distance between origin o to destination d.
Let the passenger's expected fare F_{expect} linearly depend on distance and a set price p for a unit of distance. $F_{expect} = \Delta(o,d) \cdot p$.
Let the price sensitivity of customers be $\frac{F_{expect}}{F_{offer}}$. It is worth noting that a price sensitivity value specific to an individual could be already included in the passenger characteristics and this is just a global representation for the whole of system.

Then the probability of a customer accepting a fare Q for a taxi ride at a fare price F_{offer} can be expressed as:

$$Q_{o\to d}(P, F_{offer}) = \frac{F_{expect}}{F_{offer}} \cdot V(P) \tag{1}$$

This formula is sufficient for the basic case of simulation. It can later be extended to take in account other determinants of demand as seen in Figure 1 and discussed in Section 2.1.3.

The relevant variables for passengers can be generated using a stochastic process. Similarly, passenger distribution within the network can be generated using a stochastic process. These processes could take in account some characteristics observed in reality such as demand variance during the day, lower passenger income in some areas resulting in lower willingness to pay, whether a trip is for pleasure or business, and others.

3.1.3 Taxis in Taxi Market

Let taxis have some variable costs VC for a unit of time directly related to driving and some of fixed costs FC for a unit of time. Examples of fixed costs are business overheads, insurance and deprecation. Examples of variable costs is fuel costs and taxi running costs. In this case drivers' wages are a fixed cost because it is assumed that the taxi is always active. For simplicity it is assumed that the variable cost depends only on the distance covered d. Then total taxi costs TC for a total time

t can be expressed as:

$$TC = t(VC \cdot d + FC)$$

Taxis have a set of available actions. When stopped at a location, they can decide to drive to another location or wait. If there is a passenger present, they can start interacting with the passenger, query the passenger for a desired destination and offer a price for travelling there.

When taxis interact with customers in reality in a market with no fare regulation, bargaining is likely to happen: passengers state a destination, taxis bid passengers a fare, and passengers can agree with it, or decline it and give a countering bid or abandon the process. Bargaining allows taxis and passengers to agree on a mutually acceptable price. To simulate real-world behaviour, a reinforcement learning-based bargaining process could be used as described in Cli (1997).

However, sophisticated bargaining can be disregarded in the simulation if the horizon is significantly long as the agreed fares should converge, albeit at a slower rate. A simpler approach is limiting the bargaining process to a single bid which is immediately accepted or declined by the passenger based on the demand Q. To incentivise the taxi agent, each bid also has a cost in time.

3.1.4 Modelling reinforcement learning

The research problem needs to be formally defined from a reinforcement learning point of view according to the definitions in Sections 2.2.2 and 2.2.6. Maximising the profit can be chosen as the goal of the taxi agent's operation.

It can be assumed that taxi has a complete knowledge of the road network it is operating in - a realistic assumption given modern GPS navigation systems. In reality this set would be infinite but it can be simplified to a finite set. This network forms a part of the total state space. The rest of the state space is formed of the passenger origin-destination pairs i.e. some state $s = (o,d)$. The passenger demand for each of these origin-destination states is stochastic as mentioned in Section 3.1.2.

The actions that a taxi can take were discussed in Section 3.1.3. A simple reward function is as follows. For each moment of time that a taxi is active, there is an immediate negative reward based on fixed costs and salary variable costs. If the taxi travels, it suffers a negative reward based on travel variable costs. The only positive reward it gains is from passengers paying their fares.

If passengers wait at a location for a longer time, then taxis can form expectations about the possibility of encountering passengers at each location. To keep track of this uncertainty an agent would need a belief model as discussed in Section 2.2.6. This complexity can be avoided by assuming that passengers do not wait for taxis and appear at a location for a single moment in time only.

As recognised in Section 2.2.4, the simplest reinforcement learning method to implement is Q-Learner, and it has been just proven to be adequate for the minimum requirements when the complexity is reduced.

3.1.5 Benchmark

To evaluate the variable pricing approach, benchmarks measuring taxi profitability need to be established using the linear tariff approach i.e. a certain price for a unit of time and distance. When a tariff is chosen, the simulation can be run the exact same way as with a reinforcement learning, the only

Category	*Description*
Market	A fixed environment with a fixed demand for taxi services, that supports a single taxi cruising in a road network
Passengers	Passive agents that are stochastically generated at random points in the environment. Each passenger has a destination they wish to travel to, and a set of user-specified characteristics determining if they agree to pay a certain fare for the taxi ride
Taxi	An active agent that have fixed and variable costs, and that can travel in the environment. Taxi has a set of actions that it can take: offer a fare to a passenger (if one is present), wait at a location or drive to another location. If the offer is accepted, the taxi drives the passenger to their destination, but the passenger is lost if the offer was declined.
Reinforcement Learning	A Q-learner agent for the taxi that interacts with the environment, the taxi and the passengers with a goal to maximise the taxi's profit
Benchmark	The simulation additionally supports a taxi agent with a single fixed price

Table 1: Summary of Functional Requirements

difference being the agent having no say about the fare. As the agent faces the same costs in both simulations, costs need not be taken in account. The tarrif can be set to equal the passenger's expected fare price, but this is not neccessarily the equilibrium fare price.

3.2 Non-Functional Requirements

Usability. The installation process needs to be simple and fast. As the system is very specific, a graphical user interface is not required. However, users should be able to easily learn how to use the system and how to deal with common issues.

Maintainability: the system needs to be maintainable and extensible. In addition to a maintenance manual targeted at developers, the code needs to be documented. Software needs to be modular, so that components can be easily exchanged. The development process needs to be transparent so that history and context can be easily traced if need arises, for example, to fix a bug.

Data: all properties of the simulation need to be logged so that they can later be analysed. This means logging every value that has changed at each iteration.

4 Design

4.1 Methodology

4.1.1 Test-Driven Development

The author has previous satisfactory industrial experience of working with the Agile software development philosophy (Scrum framework) and following the Test- Driven Development methodology (TDD). Scrum has a lot in common with eXtreme Programming (XP) methodology (Copeland 2001), particularly its software development aspects. Agile philosophy, Scrum and XP are most useful for teams working on larger projects (Cohn 2010), so they will not be discussed here any more. Still, TDD is one of the practices recommended by Scrum (or XP) to improve software quality, and apart from most parts of Scrum (and XP) TDD is aimed at individual software developers (Beck 2000; Cohn 2010).

The TDD cycle goes as follows: automated tests are written first, followed by writing program code which is finally refactored to improve quality – this results in a complete test coverage and a regression test suite which actually enables safe refactoring. As an added benefit the automated test suite serves as documentation for the code and helps explain developers' reasoning at the time of writing software. (Beck 2000)

Studies have found that TDD is better for code quality than traditional non-TDD fashion: Williams, Maximilien, and Vouk (2003) found that software produced in IBM following TDD had 40% less bugs, and Bhat and Nagappan (2006) found a 50% increase in code quality in two different environments at Microsoft.

However, Bhat and Nagappan (2006) also notes that TDD required 15% more time upfront to write tests. Finally, Heinemeier Hansson (2014) warns about blindly following TDD and not employing other tools and practices that could aid software development.

Therefore it was chosen to follow TDD as closely as possible without compromising other aspects of successful software development. This means that the traditional Waterfall methodology phases of requirement specification, design and testing still apply to this project.

TDD is heavily focused on black-box unit testing, but does not exclude other ways of testing software. White-box unit testing was needed to test complex parts of the software such as Q-Learner. Furthermore, integration testing is needed to ensure that the system works as a whole.

The technologies used for TDD are discussed in Section 4.3.3. The implemented tests are discussed in more detail in Section 5.2 with actual examples and discussion of how the author managed TDD.

4.2 Architecture

A software overview diagram is shown in Figure 2. This section explains the diagram and how the software operates.

When running the simulation, initially some user inputs are processed by the input module. It validates that the input data (initially parsed from files or command line parameters) is correct, stopping the program and warning the user if any invalid data was found. The rest of the system is

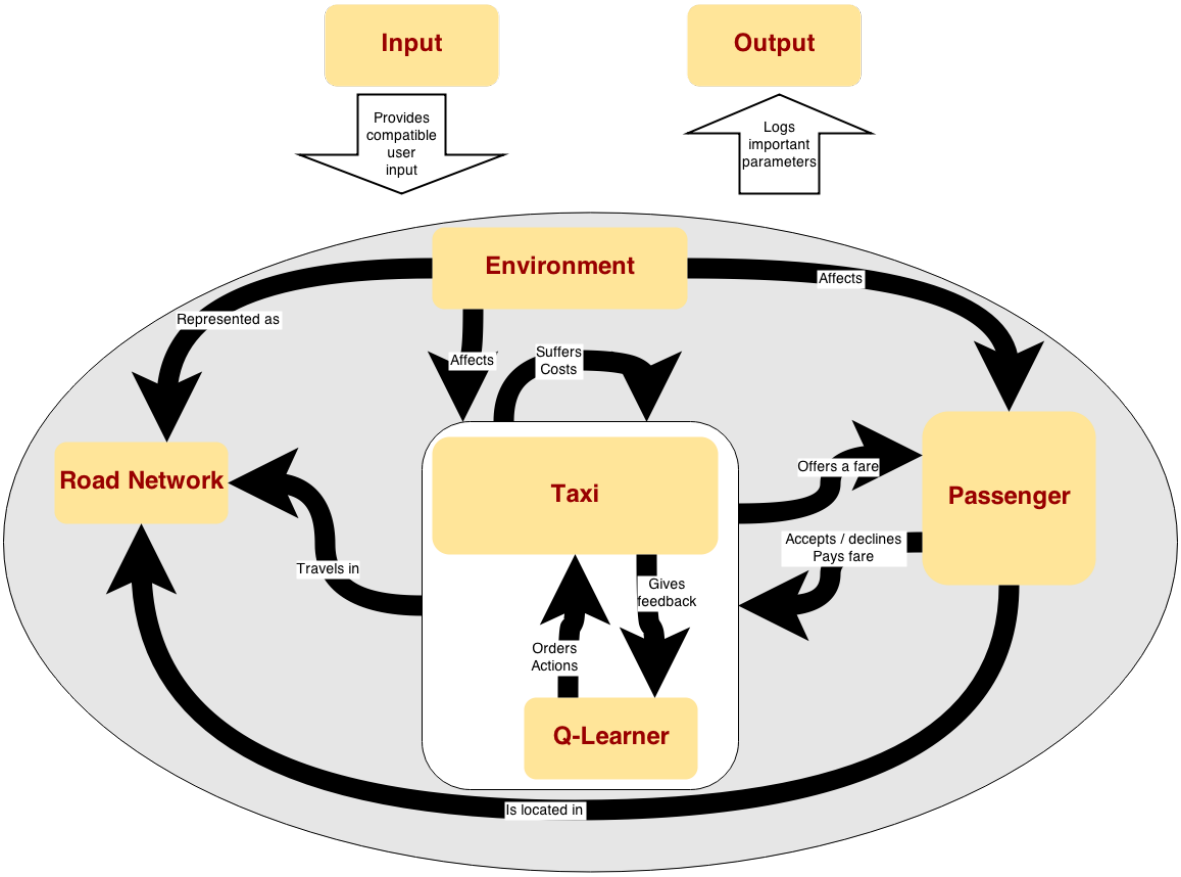

Figure 2: Software overview diagram

then initialised according to the parameters, and a simulation is started.

The environment module is the highest hierarchy simulation object where all other objects operate in. It is used for keeping track of global variables such as time and the road network. The environment can directly affect the taxi and passengers, setting their parameters, and taxis can query the environment for some information.

The road network is internally represented as a graph, and is where taxis and passengers are located and interact in.

Passengers (passive agents) can accept or decline taxis' offers respective to their internal state. If accepted, passengers travel with the taxi in the road network and pay the agreed fare. Regardless of the action taken, passengers disappear after any action is completed.

Taxi (an active agent) is the most complex object in the software. Over time taxi experiences its own costs (fixed costs and variable costs). Taxis can take three types of actions: wait at a location, drive to a different location, or offer a fare to a passenger (if one is present). Taxi's actions are decided by its learner module. The taxi passes information from the environment to the learner and receives calls to action. For example, the learner needs to know what prices it can set (from taxi itself or the environment), what locations it can drive to (from the road network), whether there is a passenger at the current location (from the environment), what the passenger's answer to the offer was (from the passenger).

Finally, there is an output module that logs simulation data to files for analysis.

4.3 Technologies

One of the core technologies used was *git* (Git 2014) version control software. Git when used in conjunction with *GitHub* (Inc. 2014) enabled easy code backups to be hosted safely online. Besides this, commit messages were aimed to be descriptive so that they serve as a form of documentation of programmer's intentions.

The rest of the technologies used for this project form distinct groups and are described in the following sections.

4.3.1 Programming Language

Ruby programming language was chosen as the core technology for this project due to author's familiarity with it and its related technologies. Ruby is an object-oriented "dynamic, open source programming language with a focus on simplicity and productivity. It has an elegant syntax that is natural to read and easy to write" (Ruby community 2014).

Ruby has many open-source libraries called *gems* available, hosted and managed locally by *rubygems* software (RubyGems.org 2014). As these gems have different versions and dependencies, they need to be managed on a per-application basis - this process is made easy by *bundler* (Bundler.io 2014). More on these tools can be found in the Maintenance Manual in Appendix B.

4.3.2 Cross-Platform Operation and Automation

Managing different Ruby versions on a single computer can be cumbersome in case it is needed for multiple Ruby applications. Furthermore, Ruby development for Windows operating systems (OS) is

```
describe Class do
  describe '#currency' do
    it 'is EURO' do
      instance = Class.new
      expect(instance.currency).to eq('EURO')
    end
  end
end
```

Figure 3: RSpec test example

several months behind development on Unix-based systems (Ruby community 2014), rendering the latest versions of Ruby unusable directly on Windows, which also happens to be the case with this project.

Fortunately, this can easily be avoided by using the following software: *VirtualBox* (Corporation 2014) virtualisation software to host any of the most popular OSs, *Vagrant* (HashiCorp 2013) to automatically manage the OSs and software installed on them, and *RVM* (Papis and Seguin 2014) to manage Ruby versions on a single OS. Detailed instructions on using this software stack with Ruby are in the Maintenance Manual in Appendix B. This method also means that the programmer can easily move between computers as the setup costs are significantly reduced by automation.

4.3.3 Supporting Test-Driven Development

As was explained in Section 4.1.1, Test-Driven Development (TDD) is an important method for software development in this project. Therefore supporting effective TDD is critical. *RSpec* is a TDD testing tool for Ruby (Marston 2014). The tests describe the desired behaviour of software in a domain-specific language aimed to be very close to natural English.

An example RSpec test scenario is shown in Figure 3. This test specifies behaviour of some ruby class 'Class' which should have a 'currency' method that returns 'EURO'. If multiple tests are written specifying the expected behaviour in different conditions, then RSpec tests become a form of programmer-friendly documentation that is executable and protects the code from introducing regression bugs.

RSpec can be enhanced by other tools. *Simplecov* (Olszowka 2014) is a code coverage tool that monitors test coverage of a codebase by checking how many lines of code are executed when running the test suite. *Guard* (Guillaume-Gentil 2014) is a tool that monitors file changes and executes appropriate events (i.e. running RSpec tests). This can make a developer's TDD work flow streamlined leaving the developer with only tests and code to write.

5 Implementation and Testing

5.1 Implementation

This section explains how the system was implemented following the software architecture specified in Figure 2 and its detailed description in Section 4.2. The software components are: User Interface (Section 5.1.1), Road Network (5.1.3), Passenger (5.1.5), Taxi (5.1.4) and Q-Learner (5.1.6). Besides a detailed look at the software modules, this section is concluded by an overview of known limitations and issues with the current implementation in Secton 5.1.7.

5.1.1 User Interface: Input and Output

User interface is a low-priority requirement and needs to provide minimal interactivity, therefore command line interface was deemed appropriate. Ruby provides many command line commands. However, to comply with a common practice for ruby applications it was decided to use *Rake* (Weirich 2014), a command-line software task management tool to make all application tasks easily available to users. A full set of instructions for using Rake and the available commands is available in the User Manual in Appendix A.2.1.

User input is provided in a text file which is parsed and instantiates the environment and other simulation objects. Some care has been taken to provide data validation to ensure a successful simulation and to give users feedback about expected common input mistakes. Unfortunately due to time constraints the data validation protection is fragmentary, so in some cases it may be harder for users to trace the wrongly input data. The full guide to using an input file is given in the User Manual in Appendix A.2.2.

Simulation output is provided in log files, important data being written with each iteration of the simulation. The output can easily be configured and changed by developers following the explanation in the Maintenance Manual in Appendix B.2.3. The output logging is a module that can be reused in other source files as needed.

5.1.2 The Environment

The environment is initialised based on the user input as discussed in the User Manual in Appendix A.2.2. The environment in turn initialises all simulation objects and agents, and runs the simulation twice: once using a range of fare prices, and once to establish a benchmark with a fixed taxi fare price. To reduce software coupling, the implementation also abstracts some methods of the road network that are used by taxis and passengers.

5.1.3 Road Network

A real world road network is represented as a weighted connected graph constructed of edges (roads) and vertices (origins/destinations). Each vertex has associated properties: a list of connected vertices with their connection distances (weights), a passenger and a taxi (if present). Taxis can travel between connected vertices, and are assumed to take the shortest routes and travel at a constant speed. Passengers appear at nodes and when taxi is at a node they can interact.

Road networks have a specific structure that is different from generic randomly generated graphs.

Eisenstat (2011) believes that the structure of the graphs is important for optimisation problems, giving the example of optimising logistics operations for a fleet of vehicles where algorithmic performance greatly differs from that on generic graphs. Uniform planar graphs are a reasonably realistic way to represent road networks (Eisenstat 2011; Masucci et al. 2009). A way to generate random planar graphs is suggested by Brinkmann and McKay (2007), including software called *plantri*.

Shortest paths in graphs can be calculated by Dijkstra's algorithm (Cormen et al. 2009). However, calculating all possible shortest paths is not necessary as some routes could never be travelled. Furthermore, that may be infeasible in a very large network as the number of paths grows exponentially. Online calculation of a shortest path between two nodes in a graph can be efficiently done using *A-star* algorithm by Hart, Nilsson, and Raphael (1968).

The network was implemented by using a graphing gem *plexus* (community 2014). An alternative graph library considered was *RGL* (Duchene 2008). Unlike RGL, Plexus had not implemented an algorithm to check for connectivity of directed graphs which could potentially be an issue. However, RGL was no longer in active development and Plexus natively supports numerous graph algorithms, including A-star shortest path search that was mentioned above.

5.1.4 Taxi

The taxi agent is arguably the most important component of the simulation. It stores its location and reachable destinations in the environment, the last reward received, a list of possible prices, and its costs. Taxi initialises the Q-Learner (see Section 5.1.6) with an initial state and a list of available actions that depend on the location, list of available prices and presence of a passenger.

A taxi is called to act by the World, when it processes an action and updates internal parameters. The Q-Learner is updated with the reward for taking the action, and the taxi is marked as engaged for as long as the action takes. Then taxi's location and total profit are updated, and it is decided if the new location has a passenger present. A new action is then chosen by the Q-Learner, which will be executed the next time the taxi is called to act.

Action is a subclass of Taxi. Each action has a known type: *wait*, *drive* to a location without transporting a passenger or *offer* a fare for transporting a passenger to a location. All of the action types have a known *fixed cost*, *variable cost* (not used for *wait*) and *units* of time the action takes. The costs and units are used to calculate a cost for action. *Drive* or *offer* actions additionally have a known destination, and *offer* action can has different costs depending on whether the offer was accepted by a passenger. When the taxi starts an action, it is marked as busy.

Each location has two states that the taxi can discern – without a passenger present or with a passenger going to a certain location. Each state is essentially a tuple of $\langle location, destination \rangle$ where the destination can be empty.

5.1.5 Passenger

As was identified in Section 3.1.2, passengers have a set of characteristics and can passively interact with taxis. As each characteristic is potentially a complex object with different values and probability distributions, it was decided to implement it as a subclass of Passenger class. Passengers agents are

generated on demand if a taxi is present, according to a location's probability of having a passenger.

These software classes were designed with the goal of eventually having user- customisable passenger details and characteristics. However, due to time pressure and higher priority goals, the passenger details and characteristics are fixed and uniform.

5.1.6 Q-Learner

As described in section 3.1.4, Q-Learner is a simple and adequate reinforcement learning method fit for the project. Of course, the very basic Q-Learner will need to be extended to accommodate exploration in this particular case of having an active agent (see Section 2.2.5). Therefore an exploration constant ε needs to be used for action selection, and it can stay fixed as the horizon for the simulation will normally be unknown.

A further addition required for the Q-Learner is a step-size function. This function is used to adjust for the bias of previous estimates. As a state is being visited more and more, any new experiences are less and less important compared to the old experiences, therefore the temporal difference weights should be adjusted less. This issue was discussed in more detail by R. Sutton and Singh (1994), where three different algorithms are introduced for optimising the step sizes for improved reinforcement learning performance. Nevertheless, the policy will still converge even if the step size simply decays (i.e $\lim_{n\to\inf} \alpha(n) = 0$ where n is the number of times a state has been visited). Therefore for simplicity the function can be fixed as $\alpha(n) = \frac{1}{n}$.

When the step size function is added to a basic Q-Learner, the algorithm adapted from R. S. Sutton and Barto (1998) can be used as shown in Algorithm 1. This algorithm updates action-state pair value estimates, and the learner can then select the optimal action depending on the exploration function (action selection algorithm is not shown here). The required inputs for initialisation and for further operation of the algorithm are clearly listed.

5.1.7 Limitations and Issues

Performance limits and issues were discussed in Section 5.2.3.

One of the main problems with the existing implementation is the non- transparent management of default variables. These variables are used for setting some parameters when user input is not supplied, and used for setting many parameters where user input is not supported yet (e.g. passenger details). At the moment these default settings are stored in their respective class source code files. This approach is infeasible because it complicates maintainability and configurability of the system. This issue could be solved by managing these values in a single file. To compensate for the loss of context, this file would need to be commented similarly to a user manual.

A less important issue is that the taxi action sequence does not mimick real world processes. In the simulation the taxi executes actions, travels and selects the next actions immediately while it is only rendered inactive for some time afterwards; this causes no problems now but could cause issues if multiple taxi agents are introduced.

User interface is primitive and requires the user to be familiar with operating a command line terminal. Additionally, there are no progress indicators shown to the user when the simulation is

Algorithm 1 Q-learning. Algorithm that needs to be called after each transition. Adapted from R. S. Sutton and Barto (1998).

Require:
s is the last state,
s' is the next state,
a is the last action,
$A(s)$ is a set of actions for a state,
R is the immediate reward received,
Q is an array hosting the current action-value estimate
H is the visit history of state-action pairs,
α is a step-size function,
γ is a discount factor.
Policy $Q(s,a)$ is initialised arbitrarily
History H is empty

1: **function** Q_LEARNING(s,a,R,a')
2: $\delta \leftarrow R + \gamma \cdot max_{a' \in A(s')} Q(s',a') - Q(s,a)$
3: $Q(s,a) \leftarrow Q(s,a) + \alpha \cdot \delta$
4: $H(s,a) \leftarrow H(s,a) + 1$
5: **return** Q,H
6: **end function**

running.

Finally, Ruby is a single threaded language, which could be problematic for a future extension to a multi-agent system.

5.2 Testing

Section 5.2.1 outlines the testing strategy and how Taxi-sim benefited from automated testing. Besides automated tests, user testing was performed when the software was nearing completion (Section 5.2.2). Finally, performance testing and the known performance limitations are discussed in Section 5.2.2.

5.2.1 Automated Tests

Test-Driven Development (TDD) was introduced in Section 4.1.1. The TDD process was not followed to the letter. Firstly, in some cases with unfamiliar and unknown problems tests and code were written intermittently as uncertainty was slowly reduced. Another reason for not following TDD was the relaxed requirements for user interface, where development alternated between writing tests first and code first while trying to find the easiest solutions to comply with requirements.

Most of the tests used are black-box unit tests. These tests allow internal refactoring of software without (theoretically) affecting the overall system functionality. As expected, black box testing was not sufficient for testing the Q-Learner where white-box testing was utilized. Additionally, the Taxi class was white-box tested to an extent, potentially suggesting that it could have been refactored in separate modules. Integration tests were used for each top-level class, for example, integration tests for Taxi class tested how it integrates Action and State classes.

At the time of submission this project's automated test coverage according to Simplecov was 99.72% (1053/1056 lines of code covered). Of course, despite the RSpec tests and Simplecov test coverage, they in no way guarantee correct and effective code and could even lead to a false sense of security. Nevertheless, it is very likely that usage of Simplecov has improved overall code quality as it did identify some code branches that had not been covered by automated tests.

5.2.2 User Testing

User testing was done when software had matured enough to have a fixed interface implementation and a basic User Manual. A computer-literate user with minimal experience in using command-line applications was tasked with using the software according to the User Manual (Appendix A). The user found no problems with the software and was able to run the simulation from the instructions alone. After suggestions from the user, example input and output files were added to the user manual, as the user found that the work flow was complicated by looking at examples in external files.

The user was unimpressed by the complex installation process and the usage of virtualisation. However, the user admitted that it was preferable if facing an alternative of possible compatibility issues.

5.2.3 Performance Testing

Only one performance issue was noticed during the usage of the system: A-star search was chosen for calculating distances between locations. However, this required repeated graph traversal in the naive form that it was implemented. This oversight was quickly mitigated by storing the calculated distances in memory and only traversing the graph for unknown distances.

The main expected software limitation is caused by the dimensionality of state- action space for Q-Learner. After a certain system-dependent thresholds it is likely to exhaust available random access memory and require writing to and reading from hard drive, thus significantly reducing simulation speed. State- action space depends on the size of the road network and the range of prices available to the taxi. To be exact, the dimensionality equals $(n-1) \cdot n \cdot p$, where n is number of locations, $n-1$ is the number of reachable locations and p is the number of available prices. Therefore the main factor limiting the performance of Taxi-sim is hardware and RAM in particular (assuming the software implementation has no unknown bugs) – this has to be kept in mind when specifying inputs for simulations.

A final less serious issue some users might run into is the considerable size of simulation log files that have a data row for each time unit of the simulation, running in hundreds and thousands of megabytes for long simulations. Managing this issue is covered in User Manual in Appendix A.

Parameter	Value
Time Limit	1 000 000
Road network	as in Figure 4
Passengers' expected fare	20
Q-Learner's greediness ε	0.9
Q-Learner's discount factor γ	0.9
Q-Learner's default value estimate	0
Taxi's variable price range	5-100 (integers)
Taxi's benchmark price	20

Table 2: Simulation inputs

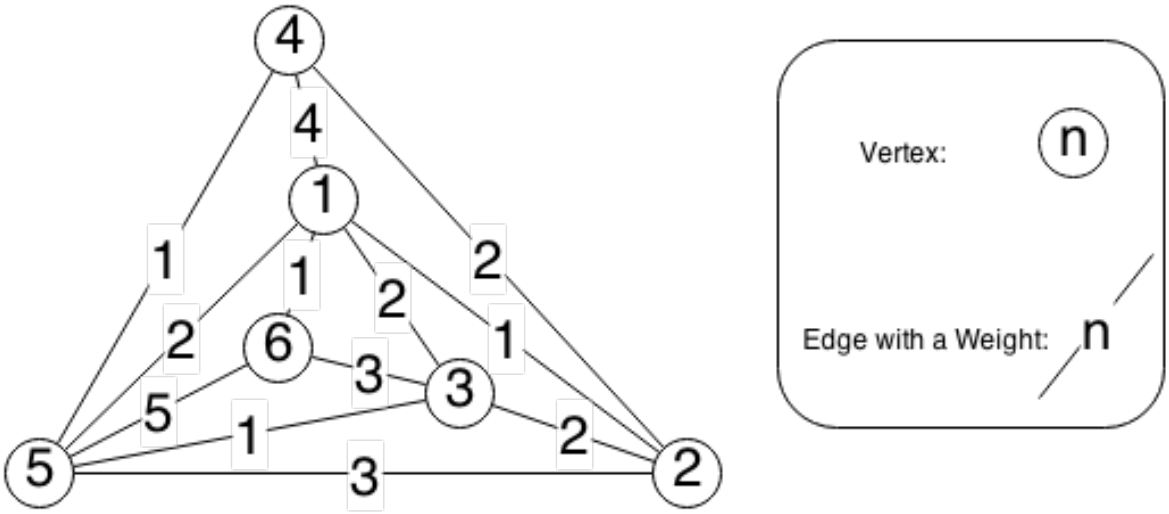

Figure 4: Simulation's Input Road Network

6 Evaluation

6.1 Simulation

When software development was nearing completion, the project's hypotheses could be tested. This section will describe how that was done, starting with explaining how the simulation was set up in Section 6.1.1. Simulation results are discussed in Section 6.1.2, with an in-depth look at the performance of the Q-Learner in Section 6.1.3. Finally, the results are reviewed in Section 6.1.4.

6.1.1 Inputs

The experiments and their analysis can be repeated by using Taxi-sim and following the instructions in User Manual in Appendix A. The result datasets can be found in the open data set mentioned in 1.1.

Initially a baseline of inputs were chosen without any intentional constraints on the environmental conditions. These inputs are shown in Table 2 and the full result data can be found in `standard` directory in the data set.

Parameter	Value	Result Directory
No changes	-	`standard`
Benchmark price	15	`lower_benchmark`
Benchmark price	30	`higher_benchmark`
Q-Learner's default value estimate	10	`value_estimate_10`
Q-Learner's default value estimate	100	`value_estimate_100`
Q-Learner's default value estimate	-10	`value_estimate_neg10`
Q-Learner's default value estimate	-100	`value_estimate_neg100`
Q-Learner's discount factor	0.4	`lower_discount`
Q-Learner's greediness	0.4	`lower_greediness`
Taxi's price range	5 - 20	`lower_range`
Taxi's price range	20 - 100	`higher_range`
Time limit	50 000	`limited_time`

Table 3: Control parameter changes

The simulation was additionally run for control purposes, varying one input parameter at a time. These runs are summarised in Table 3, displaying the parameter in question and its changed value, as well as the directory in the open data source where the results are located.

Besides the configurable inputs, some simulation parameters used defaults fixed in the source code. Passenger characteristics were manually adjusted so that the probability of a passenger accepting a fare would be about 75% for their expected fare, as in preceding test simulation runs it was experimentally determined that the formula in Section 3.1.2 resulted in this probability being lower than 10%.

6.1.2 Profitability

Full results and a detailed simulation analysis can be found included in the project's data set at the web address specified in Section 1.2. This data is organised by simulation runs as identified in Table 3, and each simulation run has two parts – variable pricing and benchmark.

Before any result analysis is started, it is important to remember that these results are relative and should be treated as such. Even though the data has exact numbers, they have no basis in reality. For example, the agents could have made a loss in all simulations had the taxi costs been set greater.

In all simulation runs, the variable pricing approach performed significantly better. The summary of total profits are shown in Figure 5, ordered descending from left to right by total profit in the simulation using variable fares.

The most profitable simulation runs were the runs with a higher fixed benchmark price and the run with default settings. The worst performance (when adjusted for the same timescale) was shown by the simulation with a limited timeframe, which most likely did not have sufficient time to find a near-optimal policy.

Interestingly, the third most profitable was the simulation where the agent was limited to a price range lower than the passenger's expected fare and lower than its corresponding fixed benchmark price. This beyond doubt proves that a flexible approach is more profitable than sticking to a fixed fare

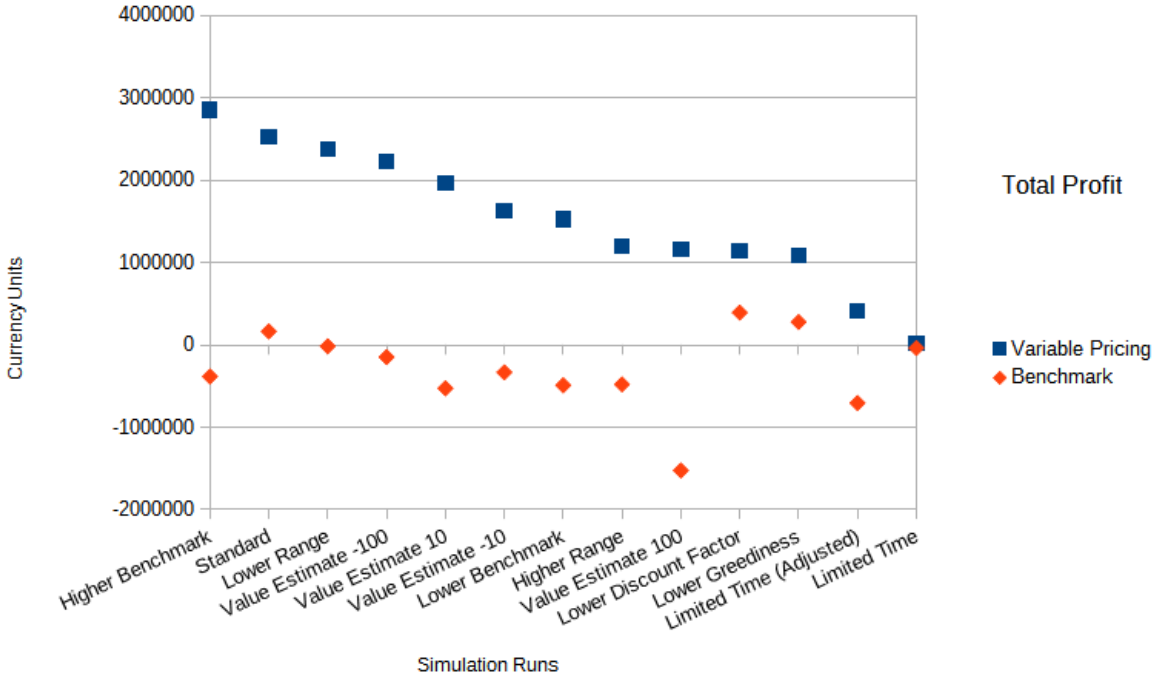

Figure 5: Summary of Total Profits in Simulation Results

price structure. However, the simulation with a higher variable price range seems to have performed badly compared to the lower price range, although it is still much more profitable than the benchmark simulation.

The effect of initial value estimates on the learner seems to have an important role when they are significantly overvalued. The simulation run 'Value Estimate 100' when comparing the variable price results fared almost as badly as the simulations with lowered discount factor and greediness that are impaired in turn by overly focusing on current rewards and by selecting a random action very often. Additionally the overvalued estimates resulted in an exceptionally low profit for the fixed price simulation. However, when the value estimates were close to the actual estimates or significantly undervalued, the simulation results show only small performance differences that could easily be explained by randomness.

Nevertheless, these results strongly suggest that in identical conditions having a choice to charge a varied range of fare prices is advantageous to a taxi's profitability, and that Q-Learning can successfully be used for setting the fare prices.

6.1.3 Performance of Reinforcement Learning

Analysis of variance was performed on each simulation's dataset (meaning that benchmark and variable pricing runs were analysed separately), and linear regression of reinforcement learning agent's observed rewards performed. The full results are available at the web address specified in Section 1.2.

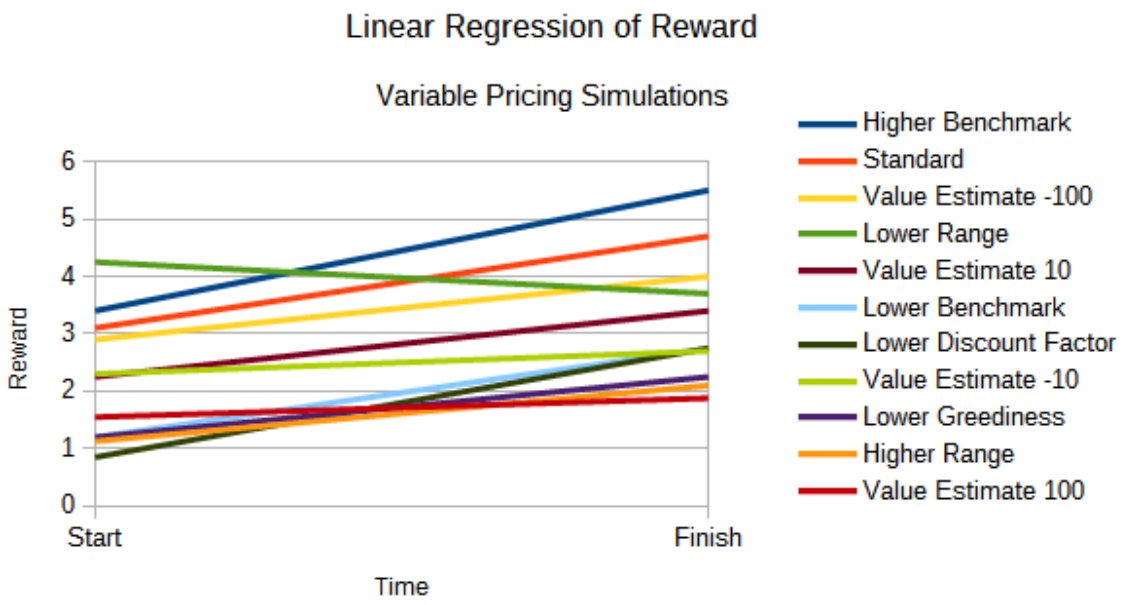

Figure 6: Regression Analysis of Rewards (Variable Pricing)

The following results have a 95% confidence rate unless mentioned otherwise.

All bar one of the simulation runs show an increase of rewards over time for variable pricing taxi agents. These reward regressions are shown in Figure 6. The agent in the simulation run with a lower range of prices might have settled on a sub- optimal policy for a large part of its lifetime, hence the trend of decreasing reward.

Data runs with a fixed price paint a similar picture (see Figure 7) with most agents steadily increasing their rewards. However, there is an outlier here as well: the simulation with a considerably overvalued initial value estimate failed to make any statistically significant change to their average reward over the course of the simulation. Unsurprisingly, this was the least profitable simulation with a fixed price as can be seen in 5 and is described in Section 6.1.2.

6.1.4 Summary

To conclude, the results suggest that the taxi agent was successful at finding a policy that increases the average reward when it can choose the price freely. However, the results clearly reveal the importance of setting the environmental conditions and constraints, which can have a large influence on the effectiveness of the reinforcement learning.

Worryingly, there was a simulation where the agent showed no change in rewards. This lack of change raises questions about the correctness of the software, as the policy was clearly sub-optimal but the rewards did not change unlike in other simulation runs. There is a possibility that this was caused by a design oversight or even a bug in the implementation, although the remainder of the runs seem to have been successful and correct.

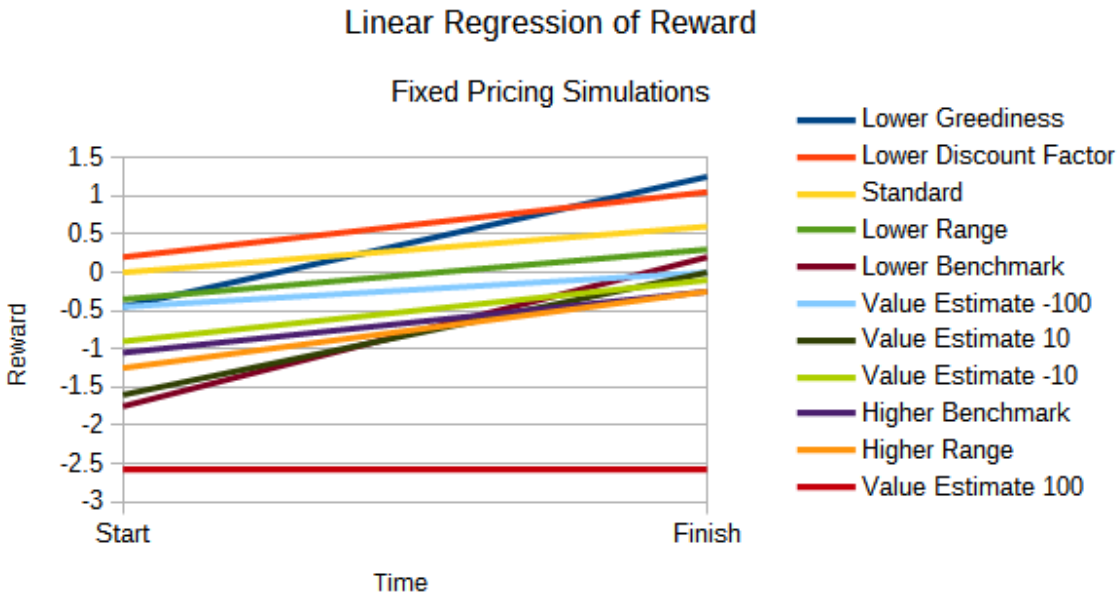

Figure 7: Regression Analysis of Rewards (Fixed Pricing)

6.2 Goals

Goals for this project were set in Section 1.3 and are discussed in the same order in the paragraphs below. The main goals were achieved to a large extent. The lower priority goals were not achieved, but have good working inroads for future.

Background. Related literature was researched in the areas of transport modelling and reinforcement learning, and comparable software was reviewed. This goal was fully achieved as it resulted in a successful piece of software that is rationalized by theory.

Software. A taxi market simulation software was successfully developed and it supports both regulated and unregulated pricing. It is grounded in the economical theory on and knowledge about modelling taxi markets. Unfortunately insufficient attention was paid to making the software to be easily configurable, therefore there are lots of important default variables that cannot be transparently changed.

Data. The simulation successfully records its data. It is relatively easy to change what data is logged. However, data analysis is somewhat complex and even requires manual data manipulation. To help users with data analysis, there is an included R language script and instructions on using it. However, the script will not work out of the box if the logged data is changed and adds more complexity to manage. Overall, achieving this data gathering-and-analysis goal required more time and effort than was initially expected. Nevertheless, the analysis revealed interesting results.

Extensibility. The software implementation is fully covered by tests and coupling has been reduced to be as low as possible. Maintainability is ensured by the modularity of software, automated

test suite that also serves as documentation, and the version control commit messages that reveal developer's intentions for code. The developer is further supported by an extensive Maintenance Manual (Appendix B). Therefore the software can be easily changed to work with new reinforcement learning strategies or changed support libraries.

Reinforcement Learning Strategy. This goal was not ever considered due to time limits. However, it is easily achievable due to software modularity as was noted above.

User Interface. This goal was only taken in account towards the end of the project, but never had a priority over the above goals. Nevertheless, the user interface proved satisfactory in user testing when used in conjunction with the User Manual (Appendix A).

6.3 Software Development

In hindsight, the software development process was successful as judged by the number of achieved goals. Still, some issues could be identified.

The automated test coverage helps to ensure that it the software is working according to specifications in the tests. However, as was noted in Section 5.2.1 it could lead to a false sense of security, which was proved true later when some unexpected bugs were discovered. When these bugs were being fixed, a few design oversights were noticed. This was caused by not revising the system architecture as the project progressed and uncertainties were being reduced; instead the development was ongoing and changes in the design were not systematically documented.

Some bugs not picked up by the automated tests were identified only when trying to use the software. This problem could have been prevented to an extent by adopting a disciplined approach to automated systems testing similar to one that was used for unit testing. This would add the difficulty of specifying system tests which admittedly would have been impossible at uncertain stages of software development.

While the final software product has low coupling, it had to be reduced intentionally. Following test driven development at times resulted in a very isolated unit-based outlook on the system, making it hard to keep the individual units in line with the whole of architecture. To improve code in terms of low coupling and other less tangible quality factors, a helpful practice can be code reviews by others – unfortunately unavailable to an useful extent in this project. From a personal point of view, working in a team is preferable to developing a solo software project as there is limited feedback one can get when working alone, resulting in a lower quality of work.

The traffic simulation software MATSim was mentioned in Section 2.3.2. It can be seen now that using MATSim could have prevented some of the modelling problems mentioned in this section and in Section 5.1. However, the performance constraints identified in Section 5.2.3 would likely severely limit the usefulness of using MATSim's micro-grained approach to traffic modelling, even if no other issues were encountered.

6.4 Future Work

The analysis of results (Section 6.1) revealed that the project has been mostly successful and the variable pricing approach could have a future. Thus it is worth to consider the future outlook of this

project. Some suggestions for developers on fixing known software issues are available in Maintenance Manual in Appendix B.2.4.

Most importantly, there were some unexpected results showing no improvement and even worsening performance of reinforcement learning (Section 6.1.3). Before any further work can be done, the cause of this contradictory result needs to be found and fixed, whether it is a software bug or a faulty algorithm implementation.

When the above issue is out of the way, further problems with the software were identified in Section 5.1. These can be fixed in the order that is deemed necessary for further experiments.

Even if the smaller issues are not fixed, the simulation can be changed to use other machine learning algorithms with minimal effort. Of course, more work would be needed to use machine learning algorithms that are not based on Markov decision processes.

Extending the simulation to work with multiple taxis is probably the most interesting future possibility from an economics position. However, the software would require bigger changes in this case as this change would implement a multi-agent system.

Passenger modelling was only partially implemented due to the time constraints, and even having good default values for passenger generation in the simulation could increase its believability when compared to reality.

If the project is looked at from a less software-centric point of view, there are other interesting considerations. The literature overview in Section 2.1 identified taxi regulation as a heated topic in transportation economics, but very few works touched the subject of the effect that deregulated prices can have on passengers and the socio-economic environment. If the variable pricing approach has any future, it needs to be considered in conjunction with its effects on the society. MATSim could be suitable simulation software for researching the effects of variable pricing on the wider economy, permitting that MATSim can be successfully adapted for this goal.

7 Conclusion

This project was mostly successful in implementing a software simulation of a taxi market where fare prices are chosen by a Q-Learner agent.

The software fulfilled all of the core goals to the minimum specification, and exceeded the minimum requirements in some areas. The software suffers from several small implementation issues that have been clearly identified, and a plan for fixing these issues was provided. While some problems were identified with the software development approach that was followed, it was adequate and proved beneficial overall.

The simulation experiment was successfully completed with twelve different scenarios. It was found that having variable fare pricing results in significantly larger profits to a taxi, confirming the project hypothesis. Furthermore, it was made clear that the Q-Learner had been a reasonable choice for controlling the taxi and it's fare pricing, as it showed a clear trend of increasing profits over time.

However, the results additionally revealed an inconclusive profitability trend in one of the intentionally challenging scenarios, suggesting that the Q-Learner was not performing correctly. Although this was an isolated instance of the problem, this leads to think that there may be a bug or a design oversight in the software which needs further investigation in future.

Nevertheless, variable pricing proved to be more profitable than fixed pricing, therefore opening future possibilities for the project's topic and software. Possible future work includes fixing the known issues with the software, experimenting with different reinforcement learning strategies or even researching the wider social implications of a variable fare pricing approach. Beyond the realm of taxis, it is worth considering to utilise variable pricing and reinforcement learning for public transport in general.

References

Bazzan, Ana LC, Kai Nagel, and Franziska Klgl (2009). "Integrating MATSim and ITSUMO for daily replanning under congestion". In: *Proceedings of the 35th Latin-American Informatics Conference, CLEI*.

Beck, Kent (2000). *Extreme Programming Explained: Embrace Change*. Addison-Wesley.

Bernhardt, K (2007). "Agent-based modeling in transportation". In: *Artificial Intelligence in Transportation* 72, pp. 72–80.

Bhat, Thirumalesh and Nachiappan Nagappan (2006). "Evaluating the efficacy of test-driven development: industrial case studies". In: *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*. ACM, pp. 356–363.

Brinkmann, Gunnar and Brendan D McKay (2007). "Fast generation of planar graphs". In: *MATCH Commun. Math. Comput. Chem* 58.2, pp. 323–357.

Bundler.io (2014). *Bundler: The best way to manage a Ruby application's gems*. URL: http://bundler.io/ (visited on 05/04/2014).

Cairns, Robert D and Catherine Liston-Heyes (1996). "Competition and regulation in the taxi industry". In: *Journal of Public Economics* 59.1, pp. 1–15.

Cli, Dave (1997). "Minimal-intelligence agents for bargaining behaviors in market-based environments". In: *Hewlett-Packard Labs Technical Reports*.

Cohn, Mike (2010). *Succeeding with Agile: Software Development Using Scrum*. Addison-Wesley.

community, Plexus (2014). *Plexus*. URL: https://github.com/chikamichi/plexus (visited on 05/04/2014).

Copeland, Lee (2001). "Extreme Programming". In: URL: http://www.computerworld.com/s/article/66192/Extreme_Programming (visited on 04/28/2014).

Cormen, Thomes H. et al. (2009). *Introduction to Algorithms*. 3rd ed. The MIT Press.

Corporation, Oracle (2014). *Oracle VM Virtualbox*. URL: https://www.virtualbox.org/ (visited on 05/04/2014).

De Vany, Arthur S (1975). "Capacity utilization under alternative regulatory restraints: an analysis of taxi markets". In: *The Journal of Political Economy*, pp. 83–94.

Douglas, George W. (1972). "Price Regulation and Optimal Service Standards: The Taxicab Industry". English. In: *Journal of Transport Economics and Policy* 6.2, pp. 116–127. ISSN: 00225258. URL: http://www.jstor.org/stable/20052271.

Duchene, Horst (2008). *Ruby Graph Library*. URL: http://rgl.rubyforge.org/rgl/index.html (visited on 05/04/2014).

Eisenstat, David (2011). "Random Road Networks: The Quadtree Model." In: *ANALCO*. SIAM, pp. 76–84.

Emele, C David et al. (2013). "Agent-driven variable pricing in flexible rural transport services". In: *Highlights on Practical Applications of Agents and Multi-Agent Systems*. Springer, pp. 24–35.

Foundation, The R (2014). *The R Project for Statistical Computing*. URL: http://www.r-project.org/ (visited on 05/08/2014).

Gao, Wenli, Michael Balmer, and Eric J Miller (2010). "Comparison of MATSim and EMME/2 on greater Toronto and Hamilton area network, Canada". In: *Transportation Research Record: Journal of the Transportation Research Board* 2197.1, pp. 118–128.

Gaunt, Clive (1995). "The impact of taxi deregulation on small urban areas: some New Zealand evidence". In: *Transport Policy* 2.4, pp. 257–262.

Git (2014). *Git*. URL: http://git-scm.com/ (visited on 05/04/2014).

Guillaume-Gentil, Thibaud (2014). *Guard*. URL: https://github.com/guard/guard (visited on 05/04/2014).

Hart, Peter E, Nils J Nilsson, and Bertram Raphael (1968). "A formal basis for the heuristic determination of minimum cost paths". In: *Systems Science and Cybernetics, IEEE Transactions on* 4.2, pp. 100–107.

HashiCorp (2013). *Vagrant*. URL: http://www.vagrantup.com/ (visited on 05/04/2014).

Heinemeier Hansson, David (2014). *Test-induced design damage*. URL: http://david.heinemeierhansson.com/2014/test-induced-design-damage.html (visited on 04/31/2014).

Ho, Don (2011). *Notepad++*. URL: http://notepad-plus-plus.org/ (visited on 05/06/2014).

Inc., GitHub (2014). *GitHub*. URL: https://github.com/ (visited on 05/04/2014).

INRO (2014). *INRO ¿ Emme transportation forecasting software*. URL: http://www.inro.ca/en/products/emme/ (visited on 04/30/2014).

Lazaric, Alessandro, Marcello Restelli, and Andrea Bonarini (2008). "Reinforcement learning in continuous action spaces through sequential Monte Carlo methods". In: *Proc. Adv. Neural Inf. Process. Syst* 20, pp. 833–840.

Manski, Charles F and J David Wright (1967). *Nature of equilibrium in the market for taxi services*. Tech. rep.

Marston, Myron (2014). *Rspec*. URL: https://relishapp.com/rspec (visited on 05/04/2014).

Masucci, AP et al. (2009). "Random planar graphs and the London street network". In: *The European Physical Journal B* 71.2, pp. 259–271.

matsim.org (2012). *MATSim: Multi-Agent Transport Simulation*. (Visited on 04/30/2014).

Neumann, Andreas and Michael Balmer (2011). *Micro meets macro: A combined approach for a large-scale, agent-based, multi-modal and dynamic transport model for Berlin*. Tech. rep. VSP Working Paper 11-14, TU Berlin, Transport Systems Planning and Transport Telematics, see www. vsp. tu-berlin. de/publications.

Noria, Xavier (2014). *A virtual machine for Ruby on Rails core development*. URL: https://github.com/rails/rails-dev-box (visited on 05/05/2014).

OECD (2007). "Taxi Services: Competition and Regulation". In: *Policy Roundtables. Competition Law & Policy*. URL: http://www.oecd.org/regreform/sectors/41472612.pdf.

Olszowka, Christoph (2014). *Simplecov*. URL: https://github.com/colszowka/simplecov (visited on 05/04/2014).

Papis, Michal and Wayne E. Seguin (2014). *RVM - Ruby Version Manager*. URL: http://rvm.io/ (visited on 05/04/2014).

Pertman, Jarmo (2013). *A wonderfully simple way to load Ruby code*. URL: https://github.com/jarmo/require%5C_all (visited on 05/06/2014).

Rubinstein, Ariel (1982). "Perfect equilibrium in a bargaining model". In: *Econometrica* 50.1, pp. 97–109.

Ruby community, Members of the (2014). *Ruby Programming Language*. URL: https://www.ruby-lang.org/ (visited on 05/04/2014).

RubyGems.org (2014). *RubyGems.org — your community gem host*. URL: http://rubygems.org/ (visited on 05/04/2014).

Russell, Stuart Jonathan and Peter Norvig (2010). *Artificial intelligence: a modern approach*. 3rd ed. Prentice Hall.

Salanova, Josep Maria et al. (2011). "A review of the modeling of taxi services". In: *Procedia-Social and Behavioral Sciences* 20, pp. 150–161.

Silva, Bruno Castro da et al. (2006). "ITSUMO: an intelligent transportation system for urban mobility". In: *Innovative Internet Community Systems*. Springer, pp. 224–235.

Small, Kenneth A and Erik T Verhoef (2007). *The economics of urban transportation*. Routledge.

Smith, Laron et al. (1995). *TRANSIMS: Transportation analysis and simulation system*. Tech. rep. Los Alamos National Lab., NM (United States).

Spaan, Matthijs T.J. (2012). "Partially Observable Markov Decision Processes". In: *Reinforcement Learning*. Ed. by Marco Wiering and Martijn van Otterlo. Springer, pp. 387–414.

Stephenson, Sam (2014). *Groom your apps Ruby environment with rbenv*. URL: https://github.com/sstephenson/rbenv (visited on 05/06/2014).

Sutton, Richard S and Andrew G Barto (1998). *Reinforcement learning: An introduction*. MIT press.

Sutton, Richard and Satinder P. Singh (1994). "On Step-Size and Bias in Temporal-Difference Learning". In: *Center for Systems Science, Yale University*, pp. 91–96.

Szepesvri, Csaba (2010). "Algorithms for reinforcement learning". In: *Synthesis Lectures on Artificial Intelligence and Machine Learning* 4.1, pp. 1–103.

Weirich, Jim (2014). *Rake – Ruby Make*. URL: https://github.com/jimweirich/rake (visited on 05/04/2014).

Williams, Laurie, E Michael Maximilien, and Mladen Vouk (2003). "Test-driven development as a defect-reduction practice". In: *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*. IEEE, pp. 34–45.

Wingate, David (2012). "Predictively Defined Representations of State". In: *Reinforcement Learning*. Ed. by Marco Wiering and Martijn van Otterlo. Springer, pp. 415–439.

Wong, KI et al. (2008). “Modeling urban taxi services with multiple user classes and vehicle modes”. In: *Transportation Research Part B: Methodological* 42.10, pp. 985–1007. DOI: 10.1111/j.1468-0270.2003.00441.x.

Yang, Hai, CS Fung, et al. (2010). “Nonlinear pricing of taxi services”. In: *Transportation Research Part A: Policy and Practice* 44.5, pp. 337–348.

Yang, Hai, YanWing Lau, et al. (2000). “A macroscopic taxi model for passenger demand, taxi utilization and level of services”. English. In: *Transportation* 27.3, pp. 317–340. ISSN: 0049-4488. DOI: 10.1023/A:1005289504549. URL: http://dx.doi.org/10.1023/A:1005289504549.

Yang, Hai, Cowina WY Leung, et al. (2010). “Equilibria of bilateral taxi–customer searching and meeting on networks”. In: *Transportation Research Part B: Methodological* 44.8, pp. 1067–1083.

Yang, Hai and SC Wong (1998). “A network model of urban taxi services”. In: *Transportation Research Part B: Methodological* 32.4, pp. 235–246.

Yang, Hai, Sze Chun Wong, and KI Wong (2002). “Demand–supply equilibrium of taxi services in a network under competition and regulation”. In: *Transportation Research Part B: Methodological* 36.9, pp. 799–819.

A User Manual

Taxi-sim is a piece software for evaluation of variable fare pricing in unregulated taxi markets. It simulates the performance of taxi agents in some environment and establishes a fixed fare taxi pricing benchmark for reference. Usage of this software requires basic familiarity with command line terminal.

This User Manual starts with installation instructions in Section A.1. Instructions for operating the software are in Section A.2.

A.1 Installation

If you do not have the source code of the software available, it can be downloaded as a *zip* archive from `https://github.com/vencha90/taxi-sim/archive/master.zip`.

For running the software it is recommended that you use virtualisation as described in Section A.1.2. If you wish to install the software stack to your operating system natively, please see Maintenance Manual in Appendix B.1.1.

A.1.1 Requirements

The software is recommended to run on a computer with at least 1 GB of RAM. All of the installation steps are compulsory unless stated otherwise. A fast internet connection could be required for installation of some supporting software, and as it relies on external services a 100% availability cannot be guaranteed.

Operating systems that are supported are: Mac OS X, Windows, Debian / Ubuntu, CentOs / RedHat / Fedora.

A.1.2 Virtualisation

Virtualisation is recommended for most Unix-like operating systems (including Mac OS X and Linux derivatives), but absolutely required for Windows. At the time of writing, Windows does not support Ruby 2.1.1 which is required for this software.

Vagrant's configuration files `Vagrantfile` and `vagrant_manifest.pp` used here are adapted from the work of Noria (2014).

- *(web)* Install VirtualBox version 4.3+ (Corporation 2014) from `https://www.virtualbox.org/` using the instructions on the website.
- *(web)* Install Vagrant version 1.5+ (HashiCorp 2013) from `http://www.vagrantup.com/` following instructions on the website.
- *(cmd)* In the taxi-sim directory, run `vagrant up`. This will provision and start the virtual machine. It will take slightly longer the first time as the operating system image needs to be downloaded.
- *(cmd)* `vagrant ssh` will connect to the virtual machine using a secure shell connection, giving you control of the virtual machine's command line terminal.

- *(cmd)* `cd /vagrant/` will change the working directory to the software. Vagrant synchronises this directory with the software directory you cloned to your machine a few steps ago.

- *(cmd)* The final step before you can use Taxi-sim is installing the *gem* (software library) dependencies by running `bundle install --local`. When this step is finished you are ready to use Taxi-sim!

If you can successfully `ssh` on the virtual machine but further steps are not working, please try the following steps:

- log out of the virtual machine by entering `exit`,

- delete the virtual machine by entering `vagrant destroy`,

- recreate the virtual machine by entering `vagrant up`,

- continue as normally.

A.2 Using Taxi-sim

Taxi-sim needs to be used from command line. If you are using Vagrant as recommended in Section A.1.2, in terminal from the main software directory run `vagrant up && vagrant ssh && cd /vagrant/ && bundle install --local && bundle exec rspec` to check that everything is in order.

You might notice that `bundle exec` is used at the start of most commands: this ensures that the commands are executed by bundler (dependency manager software) using the correct gem (software library) versions as a system could have multiple gem versions installed.

This section covers usage of Taxi-sim. To find out how to operate the software from command line terminal, see Section A.2.1. For configuring the simulation inputs see Section A.2.2. You can find out about the structure of output files in Section A.2.3. An easy way to analyse the output data is suggested in Section A.2.4.

A.2.1 Interface

Taxi-sim supports a simple command line interface. You can see the supported tasks and a short description by running `bundle exec rake -T`.

`bundle exec rake spec` runs the automated test suite and is useful for verifying the integrity of the software, especially if you have made any changes.

`bundle exec rake run[path/to/file]` is the main command for you as a user. You need to specify a path to an input file in the square brackets relative to the working directory, e.g. `rake run[sample_input.yml]`. The accepted file format is YML. Open `sample_input.yml` in a text editor to see a sample input file, a detailed explanation of the accepted inputs is in Section A.2.2.

```
time_limit: 1000000
graph:
  - "0, 1, 2, 4, 2, 1"
  - "1, 0, 2, 2, 3, 0"
  - "2, 2, 0, 0, 1, 3"
  - "4, 2, 0, 0, 1, 0"
  - "2, 3, 1, 1, 0, 5"
  - "1, 0, 3, 0, 5, 0"
passenger:
  price: 20
taxi:
  prices: '5..100'
  benchmark_price: 20
```

Figure 8: Sample Input File

A.2.2 Inputs

All types of inputs supported by the simulation are used in `sample_input.yml` file, their order does not matter. They are explained in the following list in alphabetical order:

- *graph*: an adjacency matrix representation of a road network. It has to be a connected undirected graph, meaning that the matrix is symmetrical! Self-loops are ignored. The accepted format for each row is - *"x, .., z"* where x and z are integer number edge weights.
- *passenger* is a top-level entity for passenger details and has this parameter:
  - *price* is an integer number of the price that passengers expect to pay for a single unit of distance, used to calculate probability of accepting a fare as described in Section 3.1.2.
- *taxi* is a top-level entity for taxi details and has the following parameters:
  - *benchmark_price*: an integer number of the fixed price that the simulation will be benchmarked against as described in Section 3.1.5.
  - *prices*: a range of prices that the taxi can choose from. An integer number, an array of integer numbers in the format *[x, .., z]*, or a range of integer numbers in the Ruby range format *"x..z"* (includes z) or *"x...z"* (excludes z).
- *time_limit*: an integer number for the units of time the simulation will run for

A sample input file is shown in Figure 8.

A.2.3 Output files

Simulation output is written to `logs/` directory in the application's home directory. These files can be opened using most text editing software, although some mainstream software might crash due to

```
["time", "0"]["time_limit", "1000000"]
["fc", "1"]["vc", "1"]["prices", "5..100"]
["time", "1000000"]["profit", "839289"]
["fc", "1"]["vc", "1"]["prices", "[20]"]
["time", "1000000"]["profit", "-1677056"]
```

Figure 9: Sample Simulation Summary File

```
["time", "155"]["reward", "-4"]["location", "5"]["busy_for", "3"]["action", "drive"]
["time", "156"]
["time", "157"]
["time", "158"]["passenger", "accepted"]["fare", "15"]["location", "5"]
        ["destination", "6"]["reward", "9"]["location", "6"]
        ["busy_for", "3"]["action", "offer"]
```

Figure 10: Excerpt from a Sample Simulation Log File

the large file size of `simulation.log`. In this case software that supports huge file sizes can be easily found, for example, notepad++ (Ho 2011). Parameters are logged in this format: `[‘‘name", ‘‘value"]`.

`summary.log` is a summary of the simulation results with six rows. The first row is written when the simulation starts stating the time limit, the next two rows state taxi details and the total profit for the taxi using a range of prices. The last two rows state the same data for the benchmark run with a fixed price. Contents of a sample summary file are shown in Figure 9.

`simulation.log` is a full log of the simulation as it progressed. It has two parts, the first being the run with a range of prices and the second run with a fixed taxi price. It has a single row for each time unit in both runs, therefore this file is likely to only be useful for data analysis. An excerpt from a simulation log is shown in Figure 10.

If you require customised output, the data written to log files can be changed by modifying just a few lines of the code. This is explained in the Maintenance Manual in Appendix B.2.3.

A.2.4 Data Analysis

The R Project provides an open-source statistical programming software package (Foundation 2014). An R language script file `analysis.R` for Taxi-sim data analysis is included in the project source code.

The first step before the script can be used is separation of benchmark data and variable pricing data from the main data source. Open `simulation.log` in a text editor and make two new files by splitting the original file in half by copy-pasting each run's data. Name the file with variable prices `variable.log` and the file with fixed price `benchmark.log`, and store them in the same directory. For the script to work, you have to delete the first and last line of `variable.log` where no actions are selected.

Open the script file in an R work environment. If you followed the step above, you only have to modify line 4. Change the path of your working directory in `setwd("your/path")`, for example, to be `setwd("c:`
`User/Documents/Taxi-sim/Data/")`

Run the script – it will take some time depending on the amount of data. This is a memory expensive operation and depends on the available RAM. When the script is finished, it will print a summary pdf file with graphs and descriptive statistics. This script uses external libraries available online, therefore an internet connection is required. It is expected that you will see warnings about removal of empty rows – the script measures reward that was empty when taxi had been busy with some actions.

B Maintenance Manual

The Maintenance Manual provides details of the software implementation. It gives an advanced overview of the system setup and installation and lists the main dependencies in Section B.1. A detailed guide to developing the software is given in Section B.2, including testing, documentation and suggested areas of further development.

To readers unfamiliar with Ruby, it is important to know that the term *gems* are Ruby software packages. The project is available online at this address: `https://github.com/vencha90/taxi-sim`. You might also have received a `tar` archive file of the software.

B.1 Installation of Development Environment

All commands in this section are supposed to be ran on your main operating system.

B.1.1 Installation Without Virtualisation

It is recommended to use virtualisation as described in User Manual in Appendix A.1.2. Virtualisation will ensure that only the correct required dependencies are installed and your operating system is not corrupted by potentially conflicting software. If you still do not wish to use virtualisation, there are alternatives for Unix-like systems.

You can install Ruby Version Manager (RVM) (Papis and Seguin 2014) from `http://rvm.io/` and then install Ruby by running `rvm install 2.1.1` in terminal. You also need to install Bundler by running `gem install bundler && bundle install --local` to manage dependencies.

Instead of RVM you could use `rbenv` (Stephenson 2014). Unfortunately RVM and rbenv are mutually exclusive and could cause conflicts with other parts of an OS. The worst option is installing Ruby completely natively as this provides no isolation whatsoever from other OS components.

B.1.2 Developer Setup

Even if you have the source code, you should use the latest version-controlled software from the official code repository at `https://github.com/vencha90/taxi-sim`.

To acquire the source code for development, please follow the instructions below. Commands that should be used in command line terminal are marked *(cmd)*, commands that require downloading software from the world wide web are marked *(web)*.

- *(web)* Install Git (Git 2014) version control from `http://git-scm.com/`.
- *(cmd)* Open a command line terminal.
- *(cmd)* Clone source code to a suitable directory on your local hard drive using this command: `git clone git@github.com:vencha90/taxi-sim.git`
- *(cmd)* Change the working directory to the project's source code: `cd taxi-sim`.

B.1.3 Dependencies

Top-level gems are listed in `Gemfile` and explained below. Some gems are grouped together and labelled as for development and testing, meaning that they are not necessary for simply using the

program. Potentially these labels could be used to release two separate software versions – standard and developer. The full list of dependencies is listed in `Gemfile.lock` and not explained as they only play a supporting role.

`require_all` (Pertman 2013) eases source code management by allows to require all source files in a directory on a single line in an OS- agnostic way (not supported by Ruby natively).

`plexus` (community 2014) is a graphing gem. It is discussed in more detail in Section 5.1.3

`thread_safe, '0.3.1'` is actually a dependency for other top-level gems, but was listed explicitly to specify a compatible release version (0.3.1).

`guard` and `guard-rspec` (Guillaume-Gentil 2014) can be used for monitoring file system events. A recommended work flow is suggested in Section B.2.1

`rake` (Weirich 2014) is a common tool used for Ruby command line applications.

`simplecov, ' > 0.7.1'` (Olszowka 2014) monitors code test coverage as discussed in Sections 4.3.3 and 5.2.1. The version number is specified due to a bug in the latest version.

`rspec` (Marston 2014) is a testing framework. It is discussed in Sections 4.3.3 and 5.2.1. Development with Rspec is discussed in Section B.2.1.

B.2 Developer's Guide

All commands in this section are supposed to be ran from a terminal in your development environment. This section explains how to work with the automated tests, where to find documentation and how to customise the software. Class and module names are used in this section, please see Section B.2.2 for reference.

B.2.1 Tests

Test-Driven Development (TDD) was followed as much as was reasonable when developing this project, and you are recommended to do the same. Please see Sections 4.1.1 and 5.2.1 to find out more about TDD.

Documentation and support is readily available online for Rspec. An example Rspec scenario was shown in Figure 3, but the simplistic language does not limit the functionality of the framework. It can even be easily extended if needed.

All Rspec scenarios can be run by entering `bundle exec rspec` in terminal, and help displayed by entering `bundle exec rspec -h`.

A typical developer's TDD work flow with Rspec would go as follows: write a test in text editor, switch context to command line, run the test in question (because running all tests would take a lot of time in a bigger codebase), switch context back...

This process can be automated by Guard, ordering it to monitor changes in test files and the linked code files. Guard can also be assigned conditions when the full test suite needs to be ran. It has already been configured for this project and can be started by entering `bundle exec guard`. Unfortunately this functionality is not available on virtual machines running on Windows. For virtual machines on other OSs, Guard and Vagrant needs configuration as shown here: https://github.com/guard/guard#-o--listen-on-option

lib/	Directory for source code files
spec/	Directory for Rspec test files
lib/file_parser.rb spec/file_parser_spec.rb	Input file processing – `FileParser` class
lib/graph/ spec/graph/	Road Network – `Graph` and `Graph::Vertex` classes
lib/logging.rb	Logging module – `Logging`
lib/passenger/ spec/passenger/	Passenger – `Passenger` and `Passenger::Characteristic` classes
lib/taxi/ spec/taxi/	Taxi representation, including the Q-Learner – `Taxi`, `Taxi::Action`, `Taxi::State` and `Taxi::Learner` classes
lib/taxi_learner.rb spec/taxi_learner_spec.rb	Main class and top-level namespace for the simulation – `Runner`
lib/world.rb spec/world_spec.rb	Environment – `World` class.

Table 4: Core Software Files

All tests are located in `specs/` directory. Please see Section B.2.2 for an overview of files and locations.

B.2.2 Documentation

Documentation for this software is in two forms. Table 4 in this section provides an overview of classes and modules with their related test files, and Table 5 lists the remainder of noteworthy files. Secondly, detailed documentation for each class is the automated Rspec tests. Additionally, Git commit history may prove useful for understanding the developer's intentions at the time of writing code.

B.2.3 Customising Output

Customised log details is one of the simplest changes to make to the system. It will not break automated tests if changes are made only to the output parameters, as tests only check that `any` output is produced.

Logs are currently being written from World, Taxi and Passenger classes and are handled by Logger module. To change the variables being written, you simply have to include them in `log_params` method.

Logger module can easily be included in other classes as needed, and variables to output can be specified just like in World, Taxi or Passenger classes.

B.2.4 Future Work

Some issues with the software were identified in Section 5.1.7. This section will give suggest how they could be fixed.

To transparently manage default values, they can be stored in a single `.yml` configuration file and retrieved when necessary. YAML parsing is already included in the system in FileParser class, so the

analysis.R	R language script for data analysis
bin/taxi_learner	Executable file for shell
.gitignore	File list for Git to ignore from version control
spec/fixtures/	Fixture files for automated tests
Guardfile	Specifications for Guard runtime
Gemfile Gemfile.lock	Bundler gem specifications
logs/	Log files from simulation – `simulation.log` and `summary.log`
Rakefile	Specifications for Rake tasks
.rspec spec/spec_helper.rb	Rspec configuration and support
Vagrantfile vagrant_manifest.pp	Vagrant configuration
.vagrant	Vagrant-created files for virtualisation
vendor/	Gems stored for offline usage of the software

Table 5: Supporting Software Files

hardest part of this task would be referencing the old variables to the new file. It is easier to identify the old variables, as they are currently stored as ALL_CAPS constants in source code files.

Fixing the taxi action sequence is aided by automated tests. However, this task will require serious refactoring as it would likely change the Taxi class significantly, and may need to change its subclasses as well.

Probably the easiest issue to fix is providing a progress indicator to users. This can easily be done by Runner or World classes that have access to the global time.