Multi-level evaluation of autonomous vehicle using agent-based transport simulation for the case of Singapore

Sebastian Hörl

April 13, 2016

Contents

1 Introduction

- General introduction, MATSim is suitable for simulating autonomous vehicles (Boesch and Ciari, 2015) etc ...
- Sovles last-mile problem (Litman, 2014)
- Interesting previous studies (Fagnant et al., 2014; International Transport Forum, 2014)
- Progress (Silberg et al., 2013; Schoettle and Sivak, 2014)
- Singapore (Kheong and Sheun, 2014)

TODO Introduction

2 Perspectives on autonomous vehicles

When it comes to the implementation of autonomous vehicles a lot of technological questions need to be answered, ranging from high-performance image processing to fail-safe and secure artificial intelligence. On the other side the adaption of autonomous vehicles will be highly influenced by encouragement through the authorities, companies willing to provide the services and finally the customer, who needs to decide whether he is wanting to pay the money for a ride and is comfortable enough with the technology to start using autonomous vehicles.

The following chapters will give an overview about different views on AV technology and how they should be regarded when simulating autonomous taxis in an urban environment.

2.1 The policy maker

TODO The Polciy Maker

2.2 The service operator

TODO The Service Operator

2.3 The customer

TODO The Customer

3 Simulation Framework

The investigations of autonomous vehicles in this thesis will be based on the agent-based transport simulation MATSim (Horni et al., 2015). The following sections will outline how the framework works and where it can be extended to shape it towards a autonomous taxi simulation. An overview will be given on which components need to be modified and how the final transport situation will result from all the different parts that are playing together in MATSim.

3.1 Agent-based transport simulation

The approach that is used in MATSim is to simulate a population on an per-agent timestep-based level. At the beginning of a day each person in the synthetic population has an initial plan of what it is supposed to do during the day. Mainly those plans consist of two elements:

Activities have a start and an end time, as well as, depending on the respective scenario, certain constraints on when the earliest start or latest end time could be. Alternatively activities could be defined using a certain duration after which the agent needs leave the activity location. The "standard" activities are "home" (which usually is the first plan element of a day) and "work". More elaborate simulations can additionally use an arbitrary number of secondary activities.

Legs are the second type of plan elements. These describe connections between two activity locations and contain information like which mode of transport the agent will use (e.g. "car", "public transport", or "walk") and, depending on the selected mode, further data like the route that should be taken through the street network.

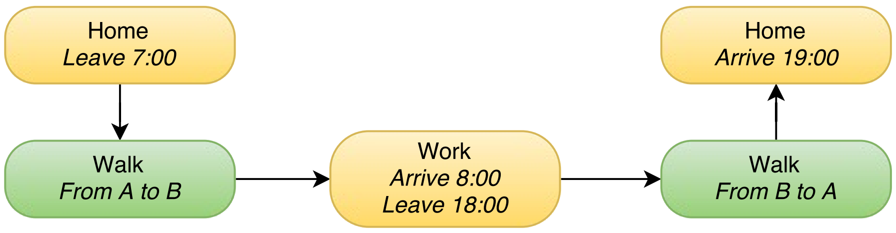

Figure 1: A typical agent plan in MATSim

A typical day plan of a MATSim agent can be seen in fig. 1. The agent starts at home, then walks to his job, stays there for a certain time and then goes back home. Each agent in a poluation (which can range from several hundreds to hundeds of thousands) has it's own individual plan that is executed when one day is simulated.

The network, on which the simulation is taking place, is decribed through nodes and connecting links. These are described by a specific capacity which tells the simulation how many vehicles are able to pass the link within a certain time frame, while the the length and an average link speed determine how fast vehicles will travel to the next node.

The whole MATSim simulation is done time-step by time-step. In a single simulation step (usually 1 second) the agents that are currently in an activity don't need to be taken into account unless their scheduled activity end is reached. The other agents, which are currently on a leg, are simulated in a dedicated traffic network simulation. This simulation moves the agents according to the capacity and current congestion from the start node to the end node of the respective links. Traffic is not simulated on a micro-level (i.e. the vehicles don't have distinct positions along the link) in order to increase the simulation speed.

The congestion on the traffic network is therefore emerging from the single travel plans of all the agents. If there are too many agents which try to use one route at the same time, the overall congestion will increase.

All steps described above, i.e. the simulation of acitivities and legs during a whole day, are summarized as the "Mobility Simulation" or, in short, Mobsim of the MATSim framework.

In order to simulate dynamic agents, which are not statically residing in a fixed-timed activity or a predefined leg route, quite significant changes need to be made and some of the computational advantages of the simulation approach need to be partly circumvented. This, however, mainly addresses implementational details and will be discussed later throughout the thesis in section 4.

3.2 Utility-based scoring

As pointed out in the last section, individual travel decisions might lead to waiting times on the network, which then can also lead to late arrivals at the designated activity locations, which usually would be disadvantageous in the real world. Therefore, it might be beneficial for an agent to reconsider the time and route choices that had been made for the current day, just as a real person would do.

In order to "know" whether a plan worked out well or was disadvantageous, the single elements need to be weighed and quantified. This is done using the Charypar-Nagel scoring function (Horni et al., 2015):

$$S_{plan} = \sum_{q=0}^{N-1} S_{act,q} + \sum_{q=0}^{N-1} S_{trav,mode(q)} \tag{1}$$

Basically, it combines the marginal utilities of all activities ($S_{act,q}$) ranging over the N activities in a plan and the marginal utilities for all the legs in between.

The marginal utility for the activities, among other factors, depends on how long the activity has been performed, whether the agent needed to wait to start it (due to an early arrival) or whether it was forced to leave early. For more details the complete computation is shown in (Horni et al., 2015).

For the scope of this thesis the travel utility is more interesting. A basic version for a single leg q can look as follows:

$$S_{trav,mode(q)} = C_{mode(q)} + \beta_{trav,mode(q)} \cdot t_{trav,q} + \gamma_{d,mode(q)} \cdot \beta_m \cdot d_{trav,q} \tag{2}$$

Mode Choice The first term, $C_{mode(q)}$, describes a constant (dis)utility for the choosing a certain mode for the leg. It can be interpreted as how "favorable" a certain mode is and is generally negative.

Travel Time The parameter $\beta_{trav,mode(q)}$ is the marginal utility of traveling, which is multiplied by the time spent on the leg $t_{trav,q}$. It signifies how favorable it is to spend time on such a leg, i.e. the longest one needs to stay in a car or in the public transport, the larger the disutility gets (and therefore the parameter is usually negative). Values which are absolutely bigger therefore stand for travel modes where time is spent less useful or comfortably.

Travel Cost The third element involves the (positive) marginal utility of money β_m, which is an universal simulation parameter and describes how the utility of money can be weighed against e.g. time. It is multiplied with the (negative) monetary distance rate $\gamma_{d,mode(q)}$, which states, to how much disutility per spanned distance the leg will lead. This parameter is useful for imposing distance-based fares in a certain transport mode and thus making it monetarily attractive or unattractive compared to other ways of traveling.

Beyond these parameters, which are usually used by all travel modes, there are a number of additions, e.g. for public transport, or yet generally unused options, such as a direct marginal utility of distance travelled.

For the purpose of simulation autonomous taxi services, two additions are made:

$$S_{av} = C + \beta_{trav} \cdot t_{trav} + \gamma_d \cdot \beta_m \cdot d_{trav} + \beta_{wait} \cdot t_{wait} + f_m(d_{trav}, t_{trav}) \tag{3}$$

Waiting Time Here, β_{wait} is the marginal utility of waiting time, quantifying how disadvantageous it is to wait for a taxi to arrive.

Service Cost Furthermore the function $f_m(\cdot)$ is a placeholder for any pricing strategy that might be tested (on top or as a replacement for the beforementioned travel cost). The implementation will allow for an easy modification of this function.

TODO Explain double penalty for waiting time due to less time to do activities

All these utility computations are done after all agents have been simulated for a whole day (in fact, usually 30h are used). This step in the MATSim framework is simply called the "scoring" phase.

3.3 Evolutionary replanning

The last step in order to make all the agents "learn" more optimal plans which make sure that they arrive on time at the activity locations and avoid congestion is to make them replan their day. This is done using an evolutionary algorithm in the following way:

Usually an agent will start out with one quite random plan, go through it during one whole day and then get a score for it. Afterwards, in some iterations, this plan is copied and modified slightly. Those modifications can happen with respect to start and end times of activities, mode choices for certain legs, etc. So after one iteration the agent might already have two plans to choose from.

Before the next day starts, one of the available plans is selected due to a certain strategy. The standard approach is to do a multinomial selection with respect to the previously obtained plan scores. So one after another plans, will be created, scored, modified, rescored and so on. Because of the selection process, which favors high scores, better and better choices will be made.

However, this is done for each and every agent, so while improving the performance of one agent's plans, this might effect the performance of other agents negatively, which is especially true if one thinks about the example of highly congested roads due to too many agents choosing the same route. Finally though, the algorithm reach at a quasi-equilibirum, which in MATSim is usually refered to as the "relaxed state", in which the average score of the used plans stabilizes within a reasonable variance.

Each "day" that is simulated in this manner is usually called an "iteration" of the simulation. It is common to divide these iterations into two parts. The first one is the "innovation phase", where plans have a certain probability to be modified, while in the second phase innovation is turned off. This means that the agent will only choose among the present plans in his repertoire (usually around 5) and rescore them again and again, until the most favorable is selected.

All of the above is known as the "replanning" in MATSim. Putting everything together, a whole cycle in a MATSim simulation can be seen in fig. 2. Since the final traffic situation evolves from the evolutionary choice in the replaning and selection, as well as from the emergent congestion in the Mobsim, this whole cycle

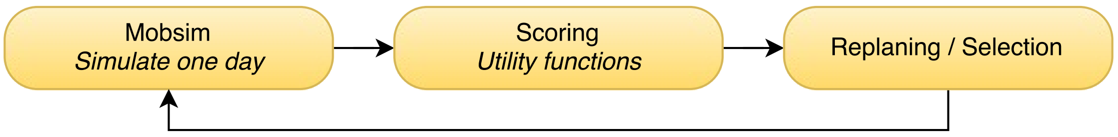

Figure 2: The basic co-evolutionary algorithm of MATSim, showing the three main stages: Mobsim, Scoring and Replanning/Selection

is usually referred to as a "co-evolutionary" algorithm.

Figure 3 shows a typical progression of the population-wide score in a MATSim simulation. What one can see there is an average over the worstor best plans of each agent, additionally the averaged average score of all the plans that the agents own is displayed. Finally one can see the average of all the plans that have been executed by the agents in a particular iteration.

The first phase until iteration 100 is the innovation phase, where time and mode choices can be done, while at iteration 100 it is turned off. There, because now the best plans must adapt to the overall situation, they are loosing in value, while the worst scored plans are dropped are likely to be discarded. Finally, a quite stable population-wide relaxed state is reached.

3.4 Queue-based simulation

The heart of the simulation in MATSim is the queue simulation, or more briefly, the QSim. This part of the framework is iterating through all the agents that need to take action in the current simulation step.

Itself the QSim is split into the handling of agents which are currently performing an activity (i.e. are in an idle state in terms of the traffic simulation) and another part, the Netsim, which is simulating the traffic network.

When running the Netsim, the agents are moved on the network according to their current route and dependent on congestion. After moving an agent, the Netsim checks whether the agent should end its trip at the current link he has been moved to. If this is the case, the agent is removed from the Netsim and the next agent state is computed.

The result of this computation is either that the agents wants to start a leg, in which case he is reinserted into the Netsim, or that he wants to start an activity,

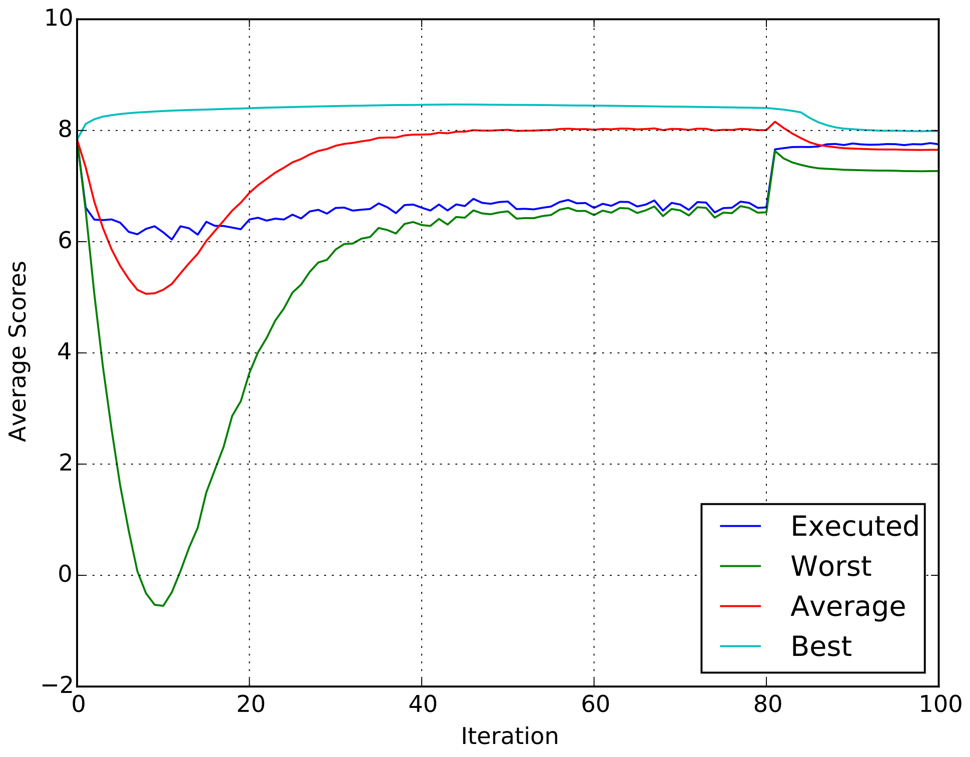

Figure 3: Typical progression of the agent scores throughout a MATSim simulation. "Worst" means that the plans with the worst scores from all the agents are taken and averaged, the same applies for the other graphs. **`TODO`** `make a better graphic with a more clear relaxation`

which will make the agent to be added to the activity queue.

This activity queue is the other main component of the QSim. In fact it is a priority queue, where all agents are sorted according to the time at which they want to end the next actvity. So when adding an agent to the activity queue it first is checked when the activity should be ended and then the agent is inserted at the corresponding position in the queue.

The processing of this queue in the QSim for each simulation step then works as follows: The first element of the queue is looked at and it is checked, whether the agent should already end the activity. If this is not the case, the simulation step is already finished. On the other hand, if the activity should be ended now (or previously if the time resolution of the simulation is quite high), the agent is removed from the top of the queue and the next state is computed as described above. Then the new top element of the queue is examined, until the simulation step is finished. The whole QSim simulation is schematically rendered in fig. 4.

The big advantage of this simulation architecture is as follows: When an agent is in an activity, no computation needs to be performed. So instead of polling all the agents in every simulation step to check if they want to end an acitivity, the computational demand is much lower when using the queue, since a lot of agents can be skipped. The sequential processing of the priority queue is very fast in terms of computational complexity ($\mathcal{O}(1)$ for checking the top element and $\mathcal{O}(\log n)$ for fetching it, Java API (2016))

Furthermore the same concept is used within the Netsim to speed up the computation of the traffic situation. In both cases, for the QSim and Netsim, those savings in computation time naturally decrease the versatiliy of the simulation environment.

One major drawback is that if an agent, which is already queued should abort its current activity, it needs to be removed from the queue and added at a new position. Both operations are quite costly for the priority queue (Java API, 2016), so if more and more reschedulings are needed, the computational overhead can become quite large.

However, for truly dynamic agents, like autonomous vehicles, it is necessary to adopt their plans frequently and thus some thought needs to be put into how to achieve this freedom while still keeping as many advantages from the existing

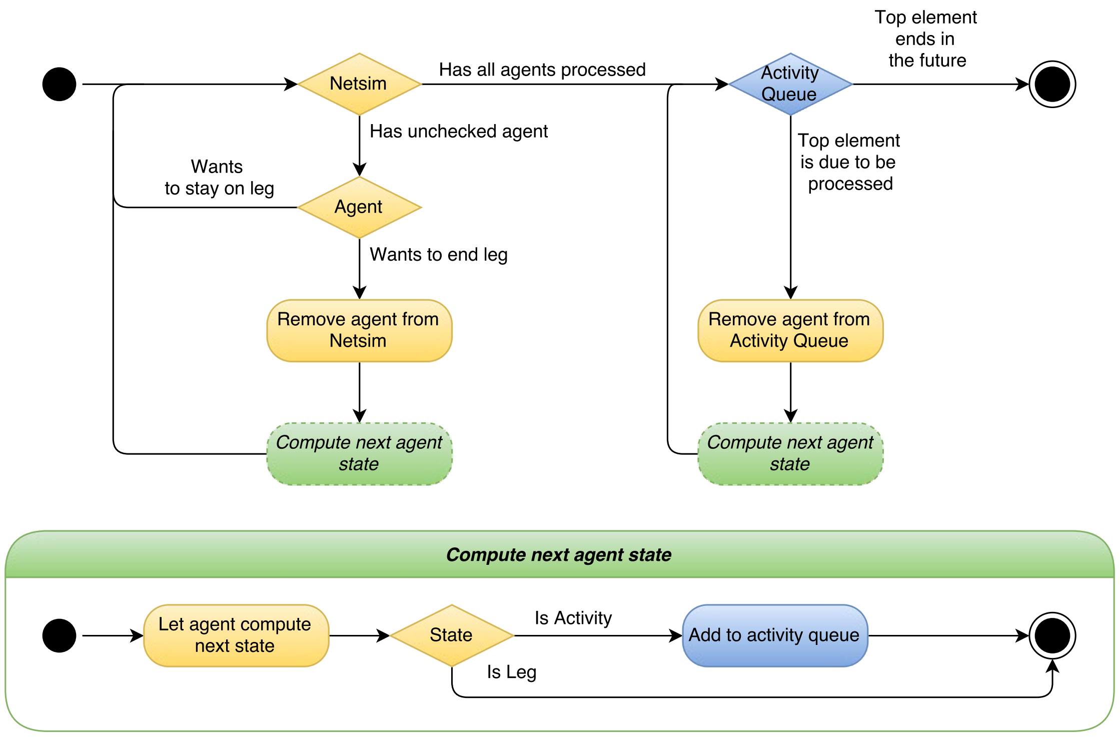

Figure 4: Simplified structure of the QSim in MATSim.

simulation architecture.

4 Dynamic Agents in MATSim

In order to allow for dynamic behaviour in MATSim, several approaches have been proposed. Mainly the decision on which option is best usually depends on what a level of complexity in the behaviour is needed.

A simple version of dynamic behaviour is implemented in the public transport extension, where passenger agents are saved in a list, as soon as they reach the link in the network, where they should be picked up by a public transport agent. As soon as the pt agent (e.g. a bus) arrives at that link, persons from the link are deregistered from the list and “teleported” to the destination stop as soon as the vehicle arrives there. This setup fits very well into the queue based structure of MATSim.

An even more dynamic approach is used in the DVRP framework, which will be discussed in the first part of this section. At the time of this thesis and for the specific use case of autonomous vehicles some drawbacks will be shown and finally a new abstraction layer for dynamcally acting agents will be presented, which has been part of the work for this thesis.

4.1 DVRP

The DVRP extension **TODO** citation is designed to provide a level of abstraction to the simulation of dynamic transport services, such as taxis. Its general structure is quite flexible, so that it is easy to implement for instance electric vehicles **TODO** citation (which need to recharge at some point during the simulation) or taxis, which are roaming randomly through the cities and serving concrete requests when they are made by passengers **TODO** citation.

However, this flexibility comes with a cost. The architecture of DVRP basically circumvents the efficient queue simulation of MATSim for activities. While "normal" agents are simulated as described before, dynamic agents (DynAgents) have two modifications:

Legs are sent to the conventional Netsim. However, agents have the ability to change their paths dynamically, i.e. one can modify the route of the agent during the simulation steps and the next time the Netsim wants to move the agent or checks whether the agent should arrive at the current link, its response is calculated from the updated path.

Activities are simulated separately from the non-dynamic agents. As soon as a DynAgent starts an activity it is added to a list of active agents. In each simulation step a specific callback for each of those agents is called and then it is checked whether the agent wants to end this activity or not. This polling approach, as depicted in fig. 5, makes sure that it is not necessary to know when an activity (for instance a taxi waiting for any requests) should end or how long it should take. On the other hand, this approach is much more computationally expensive than the efficient queue simulation. Given that the simulation is done on a second-by-second basis, an activity that would take one hour, would cost only one insert and one lookup on the acitivty queue. In the polling approach it costs 3600 calls to the simulation step callback (even if it does not actually compute anything, thise adds a computational overhead) and 3600 checks whether the activity should be ended.

So far, DVRP has been successfully applied to a number of projects **TODO** cite, though their overall usage of MATSim was different. For instance, in Bischo and Maciejewski (2016) a study on autonomous vehicles has been done, where the demand in Berlin has been obtained using an ordinary MATSim simulation. Then the link speed of the underlying network have been modified, to resemble the traffic situation in a congested situation and then *only* autonomous vehicles have been simulated. This means that the simulation could be run once and the results were obtained, because only the QSim was used and not the evolutionary learning of MATSim.

However, in the project of this thesis, autonomous vehicles should be tested in an existing multi-modal scenario, where the evolutionary learning is necessary in order for the agents to arrive at their quasi-optimal mode choices. So if Bischo and Maciejewski (2016) mentions computational times of 20h for a large number of agents, it is a measure for one iteration in the evolutionary algorithm. For the scenarios that will be tested here, a number of 100 iterations still does not reach a satisfactory equilibirum, though having a theoretical computational time of 2000h, in the case of the given paper.

Measurements have been made to determine, how much the combination of executing an empty simulation step for an agent and the execution of an activity callback that always lets the agent stay in an activity would cost in terms of execution time on the platform that has been used throughout the thesis. The final results gave an average time of 25ns. So simulating for instance 8000 agents for a common time of 30h, would lead to an execution time of:

$$T_{30h} = \underbrace{25}_{\text{Nomial}} \cdot \underbrace{3600 \cdot 30}_{\text{One day in seconds}} \cdot \underbrace{8000}_{\text{Agents}} = 21.6s \tag{4}$$

For the whole MATSim simulation, this needs to be multiplied by at least 100 iterations, so that the overhead of a simulation of 8000 agents doing nothing

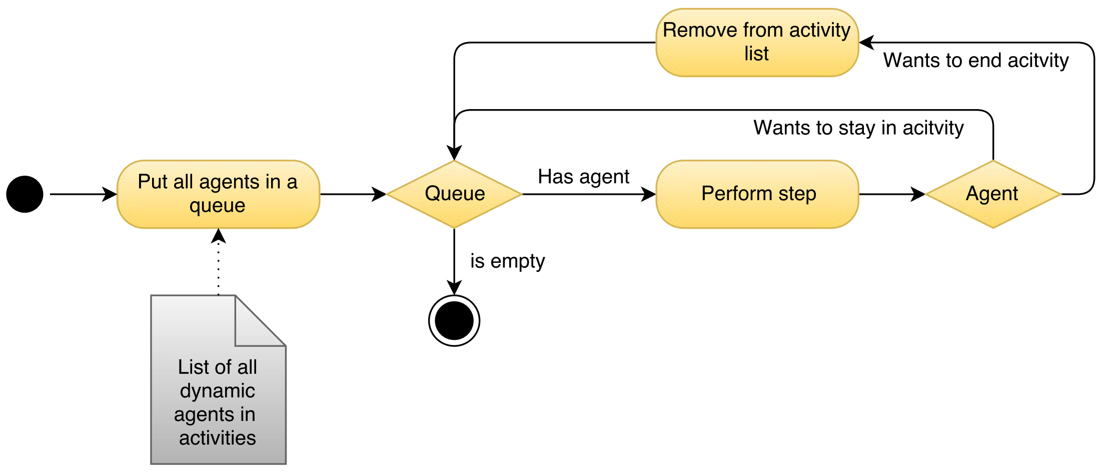
Remove from activity list
Wants to end acitvity
Wants to stay in acitvity
Put all agents in a queue
Queue
Has agent
Perform step
Agent
is empty
List of all dynamic agents in activities

Figure 5: Polling approach of the DVRP framework

would already reach a computational overhead of

$$T_{sim} = 21.6s \cdot 100 = 36min \tag{5}$$

Furthermore, while working with DVRP it has been found that the provided examples for DVRP were not fully compatible with parallelized computation in the Netsim, which would have lead to a neglect of a good way to improve the computational performance of the simulation. However, it seems like this problem has been fixed in the latest version of DVRP by the time of the writing of this thesis.

As a summary, one can say, that DVRP allows for a great freedom in simulating dynamic behaviour and its structure and signaling flows are very straightforward and easy to work with. However, for the purpose of simulating large nubmers of autonomous vehicles in the whole MATSim with large numbers of iterations, it would be beneficial to find more efficient ways for the simulation.

4.2 The AgentLock framework

As an alternative to the simulation of dynamic agents using DVRP, the AgentLock framework has been developed as part of this thesis. It tries to combine the advantages of a queue simulation while providing as much flexibility as possible to create rich dynamic agent behaviours in MATSim.

The basic idea is as follows: Usually agents in MATSim do not need a lot of processing power, i.e. the per-agent simulation step of DVRP is hardly ever used and can usually replaced by some callbacks and event handlers outside of the actual agent simulation.

Furthermore, this means that an activity basically just means that an agent is residing at a certain position in the network and not taking part of the network. So an activity for an agent is just keeping the agent back from joining the traffic network again after a certain time or event.

With this idea in mind, three different types of ways to "lock" an agent into an activity have been proposed:

Blocking activities will just let the agent reside in this activity until it is released through a call from outside.

Time-based activites are the ones from the basic MATSim simulation: They have a certain duration and therefore a fixed end time.

Event-based activities let the agent reside in the activity until a certain event happens.

The heart of the AgentLock extension is the LockEngine. Whenever it encounters an agent that wants to start a blocking or event-based activity, it is removed from the simulation and only added back again as soon as this happens manually or the event occurs. Then it either is passed to the ordinary Netsim or the next activity is starts, just as requested by the agent logic.

For time-based activities a similar approach to the activity queue for ordinary agents is taken. The first idea that would come to one's mind is to order the agents by the end of their activity and as soon as there is a change in plans, remove the element from the priority queue and add it back at a certain position.

Compared to DVRP, where a change of plans would cost nothing, here this would lead to an overhead of $\mathcal{O}(\log n)$ and $\mathcal{O}(n)$ for these operations (Java API, 2016). So it is neccessary to weigh the overhead of the polling in DVRP against the overhead of the rescheduling, which mainly depends on how often such reschedulings appear. In the concrete example of autonomous taxis, this itself depends directly on the travel demand.

TODO cite sth here for the PQ? JavaDoc?

If one sacrifices increasing memory consumption for the sake of having a faster computational time, this setup can be improved further, as done for the AgentLock framework. Here, everytime a time-based activity is started, a handle to the corresponding agent is saved into the priority queue, ordered by the end time of that activity. Furthermore an indicator is saved, whether the handle is still valid. So if in the meatime (i.e. before the end of the activity has been processed) the plan is changed, this handle will be invalidated and it will simply be ignored when processing the priority queue.

So the whole process works as follows: In every simulation step the top element of the priority queue is checked. If it is scheduled for the current or a past time step, it is processed. This means if it is still valid, the agent is "woken up" from its current activity and the next state (acitivty or leg) is computed. Afterwards, or if the handle already had been invalidated, the processing continues with the updated top element of the priority queue. This process is also shown in figure **??**.
TODO Create figure, or maybe not necessary?

The main advantage of this setup is, that elements are only removed at the top of the queue, which is significantly less expensive than removing elements at arbitrary positions within the queue.

Furthermore the AgentLock framework provides methods for dynamically ending legs and it has been made sure that all the functionality has a high degree of compatibility with the existing parts of MATSim, such as the multi-threaded Netsim.

4.3 AgentFSM

While the AgentLock extension mainly provides an abstractionl layer for the dynamic rescheduling of activities and legs, another layer has been developed, which is loosely resembling a finite state machine **TODO** cite, tailored towards agents in MATSim.

In this framework all the activities and legs are predefined states in a finite state machine and the transitions from one step to another can either be triggered manually (which is the *blocking* lock from above), by time (*time-base lock*) or by an event (*event-based lock*).

The main task of the AgentFSM framework is to encapsulate common programming steps when designing dynamic behaviour in MATSim. This means that one usually just has to define which states the behaviour is built of and how the transition from one to another works. All of this functionality is provided with a simple programming interface.

In more detail, each state is either an LegState or ActivityState. For each state an *enter* callback exists, which the programmer can use to execute arbitrary code. It needs to return how this state is locked from the three options above or either issue a direct switch to another state.

As soon as the state is ended due to the above conditions, the *leave* callback of a state is called, which must return to which next state the simulation should switch.

The concrete impementation for the autonomous vehicles will be discussed in section 5.

4.4 Comparison

An implementation of autonomous vehicles in DVRP has been compared to the implementation of the AV model developed in section 5. Basically, a specific number n of autonomous vehicles are created and put into a queue. As soon as an agent wants to start a leg using an AV, the top element of the queue will drive to that position, pick up the passenger, bring him to the final destination and drop him off. Then the AV will be added back to the end of the queue. This dispatching algorithm is quite inefficient in most of the cases, but sufficient to do investigations in terms of computational time for the two implementations, since the behaviour is equal and easily understandable.

First, simulations with $n = 2000, 4000, 8000$ have been done on the Sioux Falls scenario, which will also be described in detail in section 5. What can be seen in fig. 6 is the share of AV legs of the total number of trips and the associated computation time. For a large range of shares the implementation using AgentLock and AgentFSM (solid) is much faster than the coresponding DVRP implementation (dashed). In fact, for $n = 8000$ only at a (quite unrealistically big) share of 70%

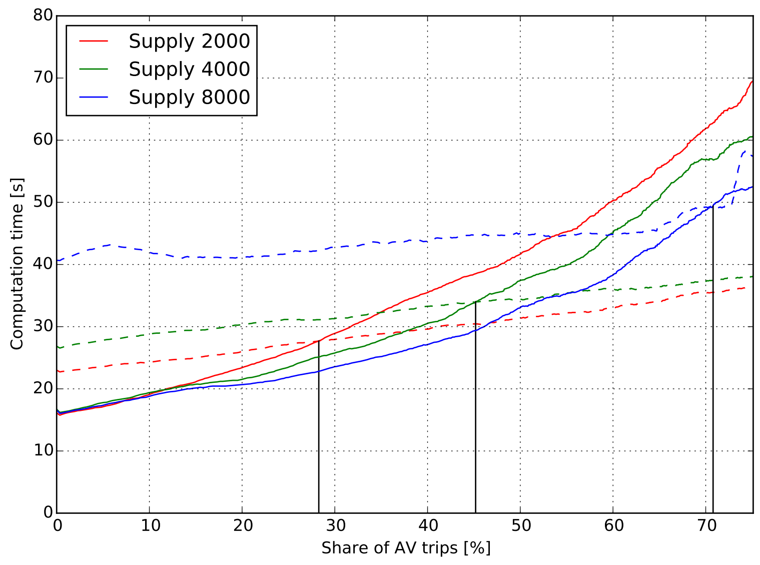

Figure 6: Comparison of Mobsim computational time between DVRP and AgentLock/AgentFSM implementation, dependent on share of AV legs. Scenario is Sioux Falls with a total number of legs of around 280.000.

DVRP starts to be more efficient.

Important to notice is that the performance of DVRP stays roughly the same over the whole range of shares. This is due to the fact that all 8000 AVs are simulated in every single time-stpe, no matter if they are active or not. In the AgentLock implementation, however, agents are only simulated if they are really in use. At 70% though, when the reschedulings of used AVs are getting to frequent, the repeated queue operations get more expensive than the polling and DVRP gets more efficient.

As a result, one can say that the FSM implementation is usually more efficient than the DVRP version. For increasing fleet sizes, this is even more true, since the roughly constant computation itme for DVRP will increase linearly with n, moving the crossing in fig. 6 further towards 100%. On the contrary, if only small fleet sizes are used, the range of leg shares with FSM in favor will get smaller and smaller, because reschedulings get to frequent compared to the number of AVs.

In terms of a multithreaded Mobsim, only tests with the new AgentLock implementation could be done, as shown in figure fig. 7. Obviously, though the

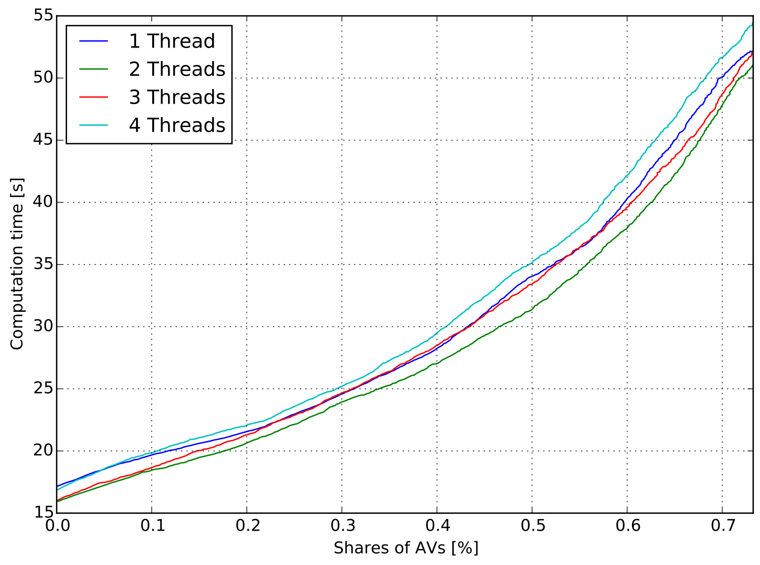

Figure 7: Comparison of different numbers of threads for the Mobsim with the AgentLock implementation

improvement is small, the simulation with two threasd performed best. This shows that it is actually an advantage to be able to use the multithreaded Mobsim. Similar results have been found for other scenarios, for instance the optimal number of threads for the Singapore scenario has been found to be four (Erath et al., 2012).

TODO For the graphics, explain better what is shown: Time for whole Mobsim; also explain how the shares are computed?

5 Autonomous Taxi Fleet Model

Simulating autonomous taxi services in MATSim requires the implementation, but even more so a concise planning and definition of how the exact behaviour of those agents should look like. This will be the first part of this section. Then, when it is known how those agents can be simulated the next question is how they should be dispatched for the requests of passengers and finally it has to be decided, where the autonomous taxis are placed on the network at the beginning of the simulation. These question will be answered in the second and third part of this section.

5.1 Agent Behaviour

The behaviour of an autonomous taxi is not a lot different from an ordinary one, except that there is no need to roam through the city to find new customers. Therefore, the AV behaviour presented here is mainly inspired by the model in **TODO** cite, where ordinary taxi services have been simulated.

The behaviour of an autonomous taxi agent can be described in terms of a task life cycle. When the agent is not in a task, he is in idle mode, basically (for the basic model) meaning that he will just stay at the current position and wait for further instructions. The following steps in the life cycle are depicted in fig. 8.

1. **Pickup Drive** is the phase where the AV has got a task and is moving to the requested pick up location. If the AV is already at the right location, this point can be skipped, otherwise it represents a leg driving from the current position to the pickup location.

2. **Waiting** is the phase in which the AV has arrived at the pickup location, but the passenger is not there yet. This can only happen if passengers request cars in advance, prior to the time when they actually want to be picked up. If the passenger is already present at the time of the arrival at the pickup location, this state can be skipped.

3. **Pickup** is the state in which the passenger is picked up. It is modelled as a fixed time, e.g. one minute and started as soon as both the AV and the passengers are present at the pickup location.

4. **Dropoff Drive** represents the leg going from the pickup location to the dropoff point.

5. **Dropoff** is the point where the passenger leaves the vehicle. Again, this is modelled as a fixed time interaction.

6. **Idle** is the final (and initial) state of the AV life cycle. At this point the AV will just wait until it receives a new task to pick up another passenger.

The states described above fit very well to the distinction of Activities and Legs in MATSim as well as to the structure of the AgentFSM framework, which has been developed for this purpose. Resulting from this behavioural model, a couple of parameters end implementational questions arise, that can be configured accordingly:

The Idle behaviour for the basic model just means that the AV will stay at its last position. Future extensions could make use of parking facilities or do an intelligent repositioning to improve the overall performance of the service.

Pickup and Dropoff duration need to be specified. In accordance with **TODO** cite, $t_{pickup} = 120s$ and $t_{dropoff} = 60s$ have been chosen. The time used for the pickup action will not be counted as waiting time (which, in turn, would be penalized through the marginal utility of waiting as described before).

The routing of the legs of the AVs is done statically, as soon as a request is created. This means that AVs will not react dynamically to the traffic situation (e.g. if there are traffic jams). For the routing through the network a plain Dijkstra algorithm is used, with the shortest path being the one with the shortest expected driving time with respect to the nomial traffic speeds on the links. This also means that the current traffic situation, which might slow down some links, is not taken into account. Those points could be future improvements of the implementation.

5.2 Dispatcher Algorithm

The dynamic dispatching of taxi vehicles is quite a complex scheduling problem, which is quite hard to solve or needs numerous asumptions to be feasible. In general, the problem will be a NP hard and heuristical solution strategies need to be applied if fast solutions are needed. An overview, as well as a proposed heuristic algorithm can be found in Maciejewski and Bischoff (2015); Bischo and Maciejewski (2016).

The algorithm can be described in two different states:

Oversupply occurs when there are more available autonomous vehicles than requests. This means that each request can be assigned without delay. In that

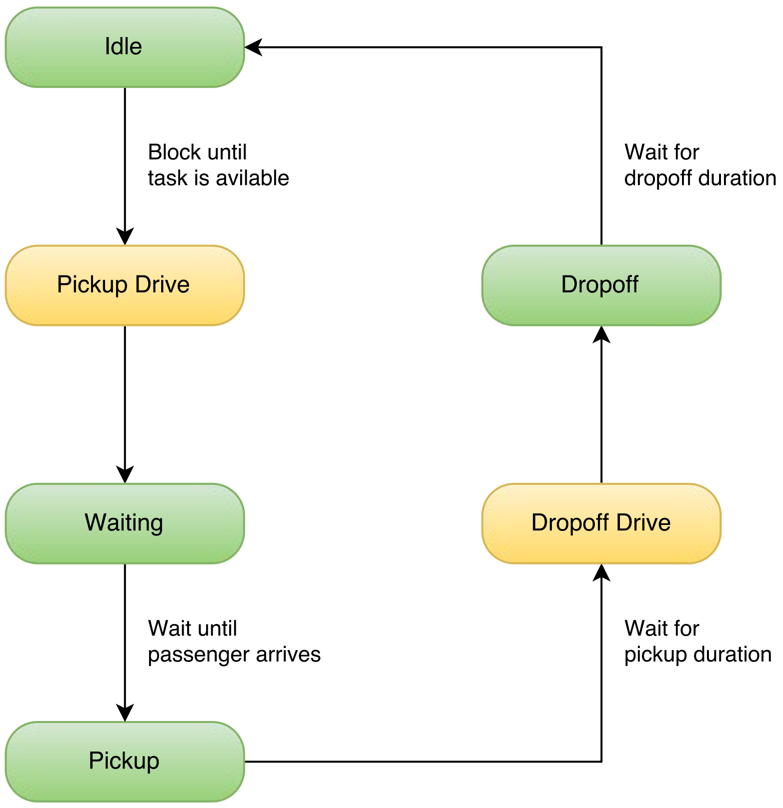

Figure 8: State chart of an AV agent. Activity states are displyed in green, while Leg states are yellow.

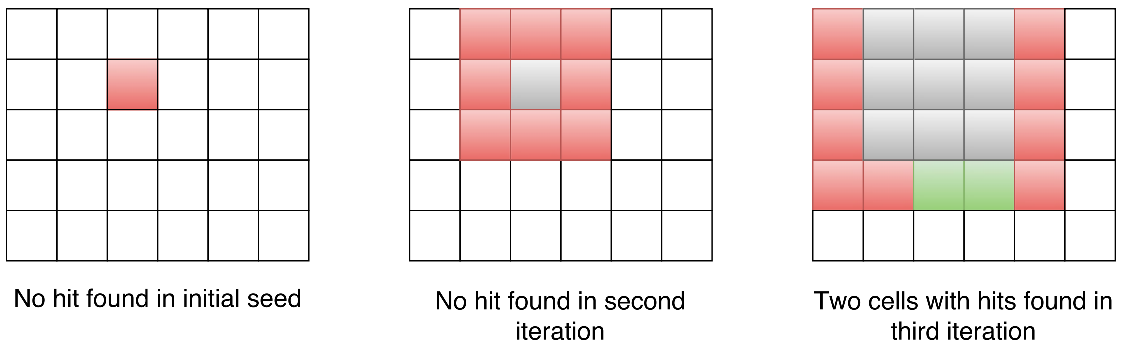
No hit found in initial seed
No hit found in second iteration
Two cells with hits found in third iteration

Figure 9: Schematic grid search algorithm for the dispatcher

case, as soon as a request arrives at the dispatcher, the closest vehicle to this request is searched and assigned to serve the customer.

Undersupply is the case when all autonomous vehicles are occupied. In that case requests will stack up, which cannot be handled immediately. In this case the algorithm works the other way round: As soon as an autonomous vehicle gets available, it is dispatched to the closest request.

According to the beforementioned papers this strategy gives a near-optimal solution, although providing fast computational times.

The implementation for this thesis is based on a spatial relaxation of the traffic network. This means that a grid with a specific resolution in x and y is fitted over the links. One of these grids saves the locations of all the available AVs, while another grid saves the locations of all open requests. Because the grids have a fixed structure, finding the closest AV or request is quite simple, since each position in x and y belongs to one specific cell of the grid. If no option is found in a certain cell, the search continues with all cells in its Moore neighborhood (all 8 surrounding cells). If this still gives no result, the radius is increased and so forth. When specifiying "find at least K hits" as a stop criterion for a search this resembles an approximate K-nearest-neighbor search based on a L_1 norm due to the grid character. The procedure is depictred in fig. 9.

This search algorithm imposes new parameters to the implementation, which basically are the cell counts in horizontal and vertical direction. Depending on the topology of the network, different parameters might be more efficient. If the grid

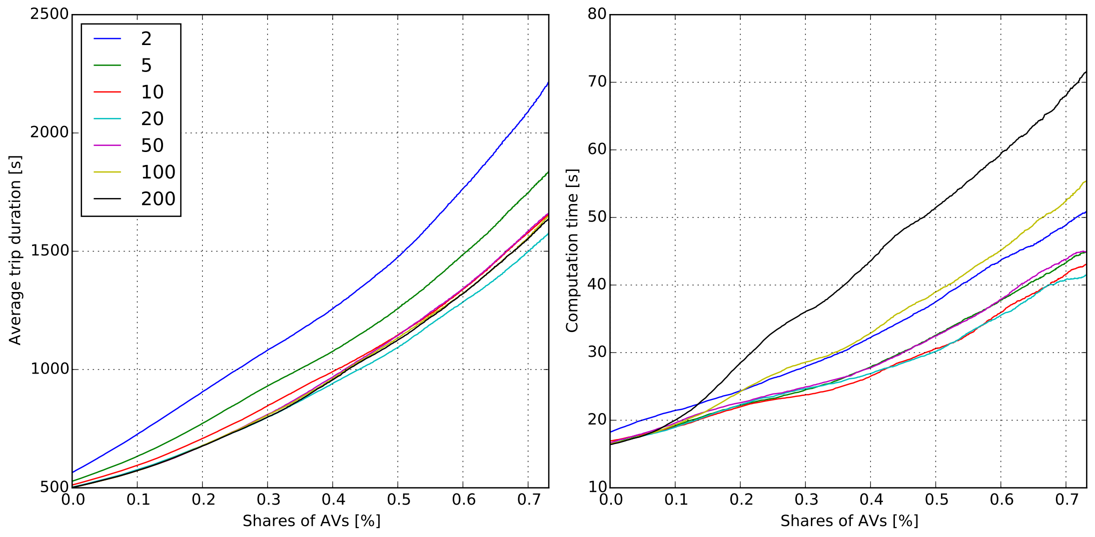

Figure 10: Performance of different grid sizes in the dispatching algorithm for Sioux Falls

is chosen to be too dense, many iterations are needed in the algorithm, while a too sparse one in the extreme case can lead to finding most of the items in only single cell.

In fact, for some topologies it might probably be more efficient to use different data structures like a binary tree or quadtree to improve the search procedure. Also more elaborate network search algorithms could be used, which make direct use of the topology. `TODO cite example maybe`

`TODO Explain how this has been parallelized! Show spoeed improvement!`

For the Sioux Falls scenario it has been tested which impact the choice of the grid size has on the results. I fig. 10 one can see that very high resolution grids ($n = 200$) lead to a great increase in computational time, due to the necessary "expansion" of unoccupied cells, as described above. On the contrary, very low grid sizes lead to very poor results in terms of the overal average travel time of the agents, because just selecting a random agent from one of a few cells is basically just random assignment. A good value for the Sioux Falls network has been found to be $n = 20$, which is computed fastest over the whole range of shares and furthermore does a good job in decreasing the overall travel time.

5.3 Distribution Algorithm

At the beginning of a day simulation MATSim all the n available AVs must be placed somewhere in the network. Two simple distribution algorithms have been implemented for the purpose of testing in this thesis.

The first one is **Random Distribution**. Here n links are chosen randomly from all available network links. While this is an easy distribution strategy for testing, it create a quite unrealistic scenario. Obviously, in a real AV service, vehicles would be relcoated over night, such that they can serve as many passengers as possible during the morning peek.

Therefore a more elaborate distribution strategy has been implemented, which should lead to more realistic results. Basically one can define a density over the network, such that every link has a certain probability to get an AV assigned. Then, in n steps, the n available AVs are added to a link dependent on the assignment probability.

This probability density can be based on many factors. The ones that are implemented for this thesis are:

Population density What is measured here is the population density in terms of how many people are havingtheir "home" activity on a certain link.

AV commuter density Here for each link it is counted how many people will have chosen the AV mode for their first leg, i.e. the one that is starting from home.

While the prior one is static, the latter on is dynamic in the sense that for every iteration in the evolutionary learning, the density will change. So while the population-density based distribution is a fixed constraint, the latter one captures the factor of availability. For instance, if there are many agents using AVs in a certain area, the availability is very high, and thus more people might be inclined to use it. On the other hand, if availability is low in an area, it might lead to longer waiting times and people are less likely to use AVs.

While it might be interesting to observe different patterns of availability, for the first testing purposes the population density is better suited, since a detailed investigation would need to be done if any results from the AV leg density come from the mode choice of the agents or of feedback behaviour within the algorithm itself.

6 The Sioux Falls Scenario

The Sioux Falls scenario for MATSim (Chakirov and Fourie, 2014) is a fictitious city with dynamic travel demand, create for the purpose of testing extensions to the MATSim framework in a realistic (i.e. using all the functionality of MATSim) setting while still providing reasonable computation times due to its rather compact size compared to real world scenarios. The model has been used in this testing for the purpose of testing the implementations and the basic functionality of the model for autonomous vehicles.

7 References

Joschka Bischo and Michal Maciejewski. Simulation of city-wide replacement of private cars with autonomous taxis in Berlin. 00, 2016.

Patrick M. Boesch and Francesco Ciari. Agent-based simulation of autonomous cars. *Proceedings of the American Control Conference*, 2015-July:2588–2592, 2015. ISSN 07431619. doi: 10.1109/ACC.2015.7171123.

Artem Chakirov and Pieter J Fourie. Enriched Sioux Falls Scenario with Dynamic And Disaggregate Demand. (March):39, 2014.

Alex Erath, Pieter Fourie, Michael van Eggermond, Sergio Ordoñez, Artem Chakirov, and Kay W. Axhausen. Large-scale agent-based transport demand model for Singapore. *13th International Conference on Travel Behaviour Research*, (June), 2012.

Daniel Fagnant, Cockrell Jr Hall, and Kara M Kockelman. The travel and environmental implications of shared autonomous vehicles, using agent-based model scenarios. pages 1–22, 2014.

Andreas Horni, Kai Nagel, and Kay W Axhausen. The Multi-Agent Transport Simulation MATSim. page 662, 2015.

International Transport Forum. Mobility : System Upgrade. Technical Report October, 2014.

Java API. PriorityQueue, 2016. URL `https://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html`.

T A N Cheon Kheong and Tham Kwang Sheun. Autonomous Vehicles, Next Stop: Singapore. *Journeys*, (November):5–11, 2014.

Todd Litman. Autonomous Vehicle Implementation Predictions: Implications for Transport Planning. *Transportation Research Board Annual Meeting*, 42 (January 2014):36–42, 2014. ISSN 10769757. doi: 10.1613/jair.301.

Michal Maciejewski and Joschka Bischoff. Large-scale Microscopic Simulation of Taxi Services. *Procedia Computer Science*, 52:358–364, 2015. ISSN 18770509. doi: 10.1016/j.procs.2015.05.107. URL `http://www.sciencedirect.com/science/article/pii/S1877050915009072`.

Brandon Schoettle and Michael Sivak. Public Opinion About Self-Driving Vehicles in China, India, Japan, The U.S., The U.K. and Australia. (UMTRI-2014-30 (October)):1–85, 2014. doi: UMTRI-2014-30.

Gary Silberg, Mitch Manassa, Kevin Everhart, Deepak Subramanian, Michael Corley, Hugh Fraser, Vivek Sinha, and Are We Ready. Self-Driving Cars: Are we Ready? *Kpmg Llp*, pages 1–36, 2013.