Large-Scale Traffic Simulation

Arundeep Singh Bahra
Student ID: 1244424
Supervisor: Dr Nick Hawes

Submitted in conformity with the requirements
for the degree of MEng Computer Science/Software Engineering with Industrial Year

School of Computer Science, University of Birmingham

April 2017

Acknowledgments

I'd like to thank my supervisor, Nick Hawes, for his invaluable support and guidance throughout the duration of this project.

I would also like to thank the friends that have made the last 5 years enjoyable and memorable, both at university and during my industrial year.

Most importantly, I would like to express my deepest gratitude to my family, for their continued support and advice during this degree and throughout my education.

Abstract

This project set out to solve the problem of simulating traffic with large amounts of vehicles on a road network. The research we performed in this project has carefully analysed the existing techniques for simulating traffic, evaluating them against each other to highlight their strengths and weaknesses in different situations. Using this information, an existing mesoscopic simulator was used as a focal point in which to develop our own requirements; these were mainly centered around usability, scalability and visualisation. The resulting traffic simulator that has been developed can simulate large quantities of cars for a user specified area of the world. It has the ability to simulate thousands of cars in a few seconds or less, and can scale to over 50000 vehicles. Many tests were run during development to analyse the output of the simulation, through which conclusions about the validity were generated.

Contents

1 Introduction

1.1 Motivation

The number of cars on our roads is increasing year by year; in 2016 alone there were over 31 million [20] in the United Kingdom. Demand for newer cars is also on the rise, with 3 million [21] newly registered vehicles coming into fruition during 2016 - most of which were purchased by private or fleet companies. There there is an growing demand to model the road network and the interaction of drivers. The introduction of smart motorways and tolls can greatly effect the flow of traffic in the surrounding area. Traffic models allow changes in road layout to be simulated in a realistic manner so that travel times and delays can be predicted. This ensures that relevant updates, such as speed limit changes, can be enforced whilst the road network is being altered.

1.2 Objective

We wanted to develop a large-scale simulator that was easy to understand and use; many of the existing simulators are either archaic in their aesthetics, or are difficult and time-consuming to get running. The term 'large-scale' describes tens of thousands of individual cars that can be put onto a road network of appropriate size. Having the flexibility to choose where to simulate was one of the main aims, as it provides a better experience for the user.

1.3 Report Outline

This report will detail the complete process that was undertaken to develop our system. The research we outline in section 2 is needed to understand the techniques we use and the terminology that will appear throughout the report. Our specification will explain how we narrowed down our project scope, taking into consideration different requirements, both from a technical and a user perspective. Furthermore, the design section explains the rationale behind technologies we chose, and how we used software engineering techniques to craft our system; this includes UML drawings ranging from high-level use case diagrams, to more detailed sequence diagrams. Our implementation was the bulk of the project; we demonstrate in this section, the challenges we faced and how we went about solving them. This includes our development methodology and testing methods that ensured robust components. The evaluation analyses the output of our system, highlighting the patterns we saw in the data, and any deficiencies that appeared. We then outline our future work that can improve the system we built, and subsequently conclude the project.

2 Research

2.1 Traffic models

Traffic simulation models have been researched for the best part of 60 years. Over this period, many methods have been derived to accurately simulate traffic and the interaction between vehicles and drivers. A large proportion of the literature as been built on top of each other, either to correct previous assumptions, or to take advantage of the increasing processing power that has become available. One of the most important aspects of simulation is the representation of vehicles and their motion; there are three methods that have become the most prevalent.

2.1.1 Macroscopic

Macroscopic simulation (Lighthill and Whitham [1] and Richards [2], Sewall [12]) represents traffic as a continuous flow. It is a technique to aggregate vehicles in terms of density at a given point on the road. Lighthill and Whitham [1] made the first significant contribution in 1955 by using kinematic waves to model the flow of traffic. Since then, it has been adapted using various techniques, such as gas dynamics.

2.1.2 Microscopic

As the name suggests, microscopic simulation ([15], [19], [17]) allows the vehicles to be modelled individually. It is the most popular method in the literature and there have a been a vast number of simulators built upon this. Microscopic simulations generally allow for more flexibility as relationships and rules between vehicles can be created to simulate more complex interactions between vehicles.

2.1.3 Hybrid

The two methods above have their advantages and disadvantages. There have been successful attempts to combine them (such as micro-macro simulation [17]) so that the benefits of both can be utilized to meet goals that could not be achieved if only one method was used. Mesoscopic (Burghout [16]) is a type of hybrid simulation and sits between microscopic and macroscopic. It aggregates the vehicles as well as having the ability to discretise them when necessary. It is a more modern approach that has gained traction in recent years.

2.2 Techniques

2.2.1 Cellular Automata

One popular technique to represent the motion of vehicles is called Cellular Automata (CA) [4]. This is a computational system that consists of a grid of cells, with each cell in a certain state. The state of each cell is updated at every discrete time step by taking into account the states of cells in its local neighbourhood [4]. This was first adopted in traffic simulation by Nagel and Schreckenberg [3]. They incorporated cellular automata in a single-lane model that consisted of an array. Each cell (at position i) can either contain a vehicle at velocity v where $0 \leq v \leq v_{max}$ or it can be empty. At each update of the simulation, four steps happen consecutively for each cell:

1. Acceleration - Velocity v of a vehicle is increased by 1 *iff* $v < v_{max}$ and the distance to the next car is greater than $v + 1$.

2. Deceleration - If the vehicle at the next occupied cell $i + j$, has $j \leq v$, then $v = j - 1$ to avoid a collision.

3. Randomisation - Velocity of each vehicle decreased by 1 with probability p. This is to simulate realistic traffic flow that occurs from human behaviour and the environment.

4. Car motion - each vehicle is moved by v cells.

The Nagel-Schreckenberg model was credited for its simplicity and efficiency. It can effectively simulate vehicles slowing down and speeding up in relation to each other. This is useful for displaying the distribution of vehicles in a traffic jam. The minimalist nature of the model, however, means that more complex scenarios can not be played out. Due to the single lane representation, the velocity of a vehicle is limited if there is another vehicle $(i + j)$ in front of it, meaning that it cannot have a velocity $j \leq v \leq v_{max}$.

There have been attempts to adapt the model to include multiple lanes. Rickert [5] extended the steps above to form a two-lane model. Additional rules are needed to allow the vehicle to realistically change lanes when another is obstructing them. Rickert applies a prior sub-step where the vehicle moves (if allowed) only sideways; the forward motion is then dealt with afterwards using the rules for the single-lane model. For a vehicle to be able to change lanes, the forward gap on its current lane and the forward and backward gaps on the adjacent lane must be sufficient. Rickert also notes the importance of stochasticity in both models; a deterministic approach in the two-lane model leads to a *ping-pong* effect, which can arise in different situations:

1. *Single lane instantiation* - if every vehicle starts in the right lane, then they all *see* a car in front of them and therefore move to the left lane (as the adjacent lane gap rules are met). The case is then repeated, this time moving to the right lane; the effect is a collective motion from side-to-side.

2. *Tailgating dance* - similar to platooning, it arises when there is a car following a leader. If the gap rule for the same lane look-ahead is set at $v+1$; and at every simulation step this is met with no other cars interfering, the follower will keep moving lanes unnecessarily. Note: this occurs only when simulating asymmetrically i.e. a vehicle always takes an opportunity to move to a particular lane.

By introducing a stochastic lane change probability check (after the gap requirements are sufficiently met), it reduces the overall number of lane changes; this is particularity effective at breaking up any platoons formed, as per (1) above. (2), however, is one of the main limitations of the model. Even when introducing a lower lane change probability, it does not resolve the issue and instead results in the follower vehicle becoming stuck behind a slower car for an unnecessary period of time. Another difficulty of this model is tuning the density (proportion of vehicles per lane) and *look-back* parameters so that a sensible flow can be achieved on both lanes. Varying density values result in different, often unrealistic, flows for each lane.

It is clear that cellular automata models at the time of [3] and Rickert [5] had both their advantages and disadvantages; although they were efficient, building a correct model was hard due to the complex parametrisation needed. Despite this, the more modern literature still uses the cellular automata approach. It is more prevalent in macroscopic classes of simulation. This is due to the grid structure allowing for an easy method to aggregate vehicles at any given time.

Sewall [12] proposes a very effective macroscopic simulation using cellular automata. Before we can describe the techniques used, we must briefly look at the historical literature of macroscopic models.

2.2.2 Continuum techniques

As mentioned in 2.1.1, [1] and [2] were the first to make a significant contribution to the macroscopic class of simulation by demonstrating the presence of shock and rarefaction waves for density and flow in traffic; this was known as the LWR model. The LWR model describes the relationship between the density ρ and flow of traffic q using first-order partial differential equations; these equations can be solved to give the density as a function of space and time. Shockwaves describe the break up of speed, flow and density; they are the boundaries of the changes in density (in space). May [6] identified six main types of shockwave, two of which are explained below to allow the reader to get a better understanding of the topic:

1. Frontal Stationary: occurs when the density is higher up the road e.g. at a red light where density is forming at the intersection. The boundary is not moving in this case.
2. Backward recovery: occurs when the boundary moves up the road e.g. when a traffic light turns green, the vehicles will move off one by one.

The LWR model had notable limitations, the main one is that it did not factor in the effect of acceleration / deceleration of vehicles. A second-order model was introduced by Whitam [7] and Payne [8] (PW model) to address this. Daganzo [9] noted that their model and most of the other second-order models were flawed due to the possibility of outputting negative velocities. Ultimately, this meant that vehicles could go against the flow of traffic. Aw and Rascle [10] and Zhang [11] solved this problem by implementing anisotropic models; their work is collectively known as the 'ARZ' model. This model is a law that can be used to emulate the motion of traffic flow; it consists of partial differential equations that describe gas dynamics.

Sewall [12] uses the ARZ model to simulate the motion of vehicles along lanes of a motorway. Although the methods used are macroscopic based, the notion of a 'carticle' is implemented; this is a discrete representation of the vehicles and it is used for visualisation and simple decision making. At each time step, there are five procedures that [12] outlines:

1. Solve ARZ equations for each lane
2. Complete lane changes
3. Advance carticles
4. Relax the relative velocity
5. Update the network as a whole

(1) needs to be solved so that the flux can be calculated for adjacent cells along a lane. In the literature this is deemed the *Riemann problem*, and it is where most of the computation is required.

(2) is achieved both at the continuum level and at the carticle level. For the underlying flow to 'change' lanes, there must be a duplication of density and velocity of the flow so that it simulates the vehicles occupying both lanes; this could require several simulation steps. Once the lane change is complete, the quantities in the old lane are removed. This lane-changing model allows the simulation of exits and entrances on a motorway.

The lanes are split into cells which contain information for more than one vehicle (a continuum velocity). The number of cells per lane is calculated using a target number for the network; the lane length is divided by this number to get the number of cells; the lane length is then divided by the number of cells to get the cell length. At each cell boundary, the updated speeds and fluxes are calculated.

2.2.3 Discrete techniques

Microscopic traffic models can simulate realistic, individual driver behaviour using complex rules and parametrisation. They were first introduced in the 1950s by Reuschel [13] and Pipes [14]. The basis of these is a car-following model where the acceleration of the follower vehicle is determined by the acceleration and velocity of the leader and the distance between them. A main microscopic simulator is MITSIM [15], and it will be our focus when describing the various techniques.

[15] does not use cellular automata; instead, it uses nodes, links, segments and lanes. These are loaded from a database file containing all objects (traffic sensors, lane connections etc.) in the road network. Nodes represent intersections or boundary (edge of the network) specifiers. They contain information regarding their type, as well as a unique identifier and a name. Links connect nodes together if their is at least one lane between them, and contain segments. Similarly to nodes, they have a type and ID attached. To switch from one link to another, a lookup table containing turning restrictions is consulted. Segments are pieces of road that group together road data; they are identified by their ID, speed limit, lane, and geometry. MITSIM is a lane-centric model and as such, requires detailed lane information, such as regulation and privilege codes. These determine if lane changes are allowed and if there are any restrictions on the types of vehicles that can use them. Lane connections are stored in table containing pairs of lane codes.

Like the model in [12], iterations are represented by discrete time steps. At each simulation step, the following flow is achieved:

1. Initialisation Phase: consists of initialising parameters, road network, origin-destination matrices and processing vehicle generation.
2. Update Phase: for *every* vehicle calculate the acceleration rate and proceed with lane-changing model.
3. Advance Phase: for *every* vehicle advance their position and update their velocity. Check for terminating criteria such as destination arrival.

In (2), the car-following model is applied. The model has several uses in addition to acceleration calculations, such as lane merges and yielding.

The image above shows the car following model from [15] with the descriptors:

- n the follower.
- $n-1$ the leader.
- L_{n-1} length of the leader, $n-1$.
- g_n distance between the leader and the follower.

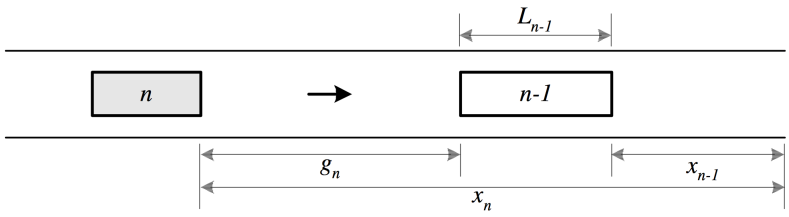

Figure 1: Car-following model. Source: [15]

- X_n is the distance between n and the downstream node.
- X_{n-1} is the distance between $n-1$ and the downstream node.

Depending on the value of g_n, a particular vehicle can be in three states: car-following, emergency state due to obstruction, or free flow. In a free flow motion, the car accelerates or decelerates to the required speed of the driver (set in parameters). In order to avoid collisions, the vehicle must decelerate at rate that guarantees the distance between it and the leader is sufficient. If the car is in car-following state, its acceleration / deceleration, a_n, is calculated using velocities, v_n and v_{n-1}, model parameters (either negative or positive depending difference between v_n and v_{n-1}), and distance g_n.

Lane changing in [15] is similar to the rules explained in 2.2.1. A car can only change lanes when suitable gaps (in front and behind) on the target lane are found. An interesting feature of microscopic models, however, is that this gap can differ between individual drivers and in different situations (for a particular driver) - the gap is calculated from traffic density and the distance to the part of road where the driver needs to have completed the lane change. The idea of 'Forced Merging' is described; cars will force themselves onto other lanes (causing others to yield) if the probability of certain criteria is met i.e distance to junction, the duration they have been waiting, and the number of changes in lane required. The gaps at an intersection are harder to set due to conflicting motions of other cars. In MITSIM, conflict points are established by extrapolating the position of conflicting vehicles. Estimations are then made so the driver can alter their speed on the intersection approach so that the distance between them is maximised; they can then make the appropriate turn without a collision.

In terms of routing, vehicles can either have a pre-set routes or have dynamic paths allocated as the journey progresses. A route choice model is applied at each intersection for those that do not have pre-planned trips; the model outputs the probability of choosing a certain link given a destination node and and estimated time of arrival. Vehicles that have pre-set routes have the ability to switch their paths *en-route*. This route-switching model takes into consideration the set of available paths from a node to the destination (via a path table file). The probability that is output from the model is used in conjunction with a random number to decide whether the path should be switched.

Parameterisation is a huge aspect of microscopic models. Individual drivers can each have separate rules (e.g. maximum acceleration / deceleration, desired speed, driver reaction time) which introduces an increase in both complexity and computation. Calibrating these parameters is difficult and there have been papers solely dedicated to this topic. It is one of the main factors why simulating large areas with microscopic techniques is not feasible.

2.2.4 Mesoscopic techniques

As per the description in 2.1.3, mesoscopic models sit between macroscopic and microscopic simulations. They have the ability to represent *simplified* vehicle behaviour and are also able to represent large networks. We will analyse the techniques used in [16] that were applied to build their mesoscopic simulator, named Mezzo. This model represents the network similarly to [15]; it uses a graph of nodes and links - nodes can represent intersections, whereas links are the 'road' part connecting nodes together.

A major difference with [16] compared with models we have previously discussed, is that it is not a time-based simulation, rather it is purely event-based. Therefore, each simulation iteration only occurs when a new event happens. Events occur when a vehicle enters and exits a link, or switches from one link to another.

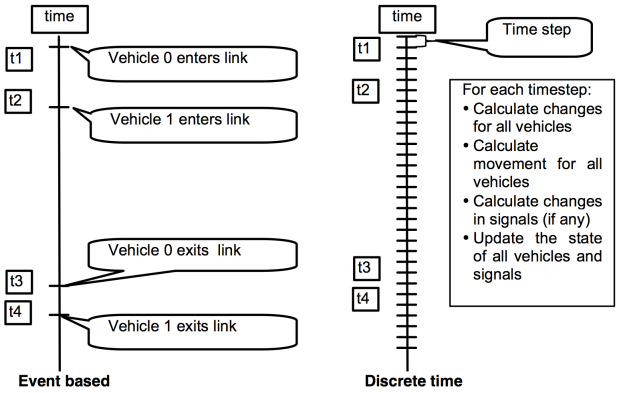

Figure 2: Event-based vs. time-based simulation. Source:[16]

Each event is generated by an 'Action', after which it gets added to an event list with a stochastically generated time interval. These time intervals are randomly drawn by a server; the event will get executed at the current time plus the time interval for that event. This event-based approach is not limited to mesoscopic models, but suits the link model described. As figure 2 shows, the reason for this is that the motion along a link is not constantly monitored, so a discrete time based simulation would waste computational effort on simulation steps where there is no state to be updated i.e. there are fewer interactions between the vehicles.

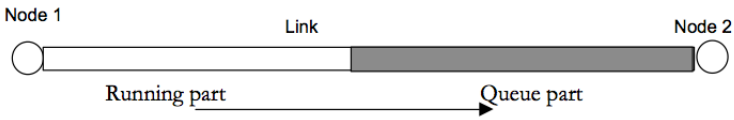

Figure 3: Link-model in Mezzo. Source:[16]

Each link is partitioned into two groups, a *running part* and a *queue part*. Depending on the flows in and out of the link, the two sections change in size to handle the changes. The running section grows towards the link exit, whilst the queue section grows towards the link entrance; representing the build of traffic along the link. Therefore, a link with no vehicles means the running section grows to be the same length of the link, and vice versa for the queue section (i.e.

a red light at a node). It is important to note that the vehicles in the running section are not effected by the build up of vehicles in the queue section until they have crossed the boundary.

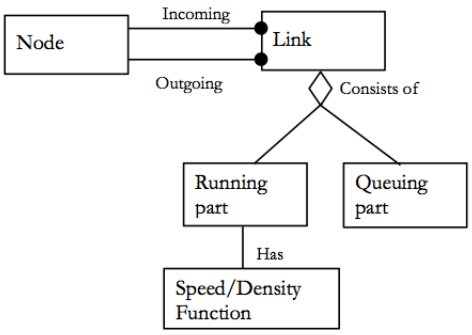

Figure 4: Road representation in Mezzo. Source:[16]

Although lanes are not represented individually, each node has an incoming and outgoing link, this acts as the separator of direction. Overtaking is simulated by comparing the outcome of the speed / density function (figure 5) of each vehicle. The outcome of the density function is described in [16] as how many vehicles per kilometre per lane on the running section only. If there is a low density then the vehicles are given a higher speed, a higher density will mean the vehicles are assigned a lower speed. The vehicles are then ordered on the link by their earliest exit time. This exit time is used to show which vehicles are on the queue; vehicles with an exit time less than an assigned time t for a particular queue. Once in the queue, a vehicles speed is determined by the motion in which the queue dissipates, this is described later. The way overtakes are handled using the speed / density function demonstrates the simplicity of motion compared to microscopic models. A vehicle a entering a link later with a higher speed is assumed to overtake one with a lower speed b, yet the queue order is not representative of the event order due to the stochastic event list; vehicle b could leave the link earlier if given a lower time interval by the server.

Link exit handling at a node is dealt with using queue servers; each connection to another link is handled by a separate server - they will pick vehicles whose route conforms to that direction. A vehicle is only processed onto a destination link if there is room on the running section of that link. The frequency at which a vehicle is served resonates with how real junctions play out: cars can only proceed when there are no cars travelling in a conflicting direction, it is at the front the queue, or there is a green light (for signalised intersections). Much like the overtaking dilemma, the mesoscopic nature of the model doesn't allow for complex vehicle behaviour; therefore, jumping the queue to turn a direction is represented by a look-back limit - a vehicle has to be in a certain position (the limit) back in the queue to be able to proceed with their route.

There have been many speed / density functions described in the literature. The simplest equations use two constants to represent the maximum density (k_{jam}) and maximum free flow speed (k_{free}), this creates a linear relationship. However, the earlier functions were found to have deficiencies, such as not accounting for a density (k) of 0. The function used in [16] is as follows:

$$V(k) = \begin{cases} V_{free} & ,if\ k < k_{\min} \\ V_{\min} + \left(V_{free} - V_{\min}\right)\left(1 - \left(\frac{k - k_{\min}}{k_{\max} - k_{\min}}\right)^{a}\right)^{b} & ,if\ k \in [k_{\min}, k_{\max}] \\ V_{\min} & ,if\ k > k_{\max} \end{cases}$$

Where

V_{min} = minimum speed

k_{min} = minimum density where speed is still a function of density

k_{max} = maximum density where speed is still a function of density

Figure 5: Speed / density function in Mezzo. Source:[16]

Queue dissipation in Mezzo follows the LWR model explained in 2.2.2. A problem with the Mezzo model is that the backward recovery (2.2.2) is not implemented properly by default. This is due to the Mezzo mechanisms; a vehicle with an early exit time could theoretically leave the link as soon as a traffic light turns green. Obviously, this doesn't represent an actual dispersion. To fix this, a new exit time is calculated for each vehicle. This is achieved by combining the time until the shockwave reaches it and the time until the downstream node.

In [16], routing is virtually identical to the techniques described in 2.2.3. There is the ability to have a pre-determined route that can be switched during the journey. The generation of routes is achieved using a combination of historical link times and a database of known routes. When generating traffic, a origin-destination (O-D) matrix is used; each O-D pair is created using a separate process - the type of vehicle (e.g. truck, car) is chosen randomly along with the chosen route (from the database). These routes are normally formed using a shortest-path algorithm, such as K-Shortest path. The downside to this is that sending all traffic for a popular O-D down the same shortest path results in congestion - the load is greater than the capacity of the links. Drivers are allowed to switch routes *en-route* if the following cases arise: a road is blocked, excessive delay arises (affects expected time), or if there is an incident on a link. If an incident occurs, a broadcast is sent out to a proportion of drivers. The broadcast information contains details of the delay, such as the link(s) in question and the expected delay. Drivers can respond accordingly if the incident is on a link on their path. Alternate routes are contemplated by comparing its travel time with the current one (including the delay).

3 Specification

3.1 Project specification

The research displays the underlying techniques of the various simulation models. Given the time limitations enforced, it would have been infeasible to attempt to deviate from these and reinvent the wheel. Therefore, a choice of the three highlighted models in 2.1 was made and the project was built upon the chosen model. The scope of the project was focused around the movement of vehicles on the road, and how increasing the individuals affected the output of the simulation. Although traffic generation is an important aspect of a traffic simulator, it was not our main focus.

3.2 Use cases

The degree of user input depends on the simulation in question; most allow the user to select a map area and build hand made roads through a GUI. The downside of this is that it can be both time consuming and complicated. Through our experience, trying to build roads in [16] was not easy and the user interface was far from intuitive. This was due to the awkward tools used to build roads; they mimic a drag-and-drop method which can become frustrating after extended use. Hand built roads also mean curved roads are hard to generate. With [16], one can also import a special file that contains the map data, including: nodes, servers, links, vehicle information and other road details. This proprietary file type is hard to setup and the visualisation output is not the most aesthetic, as the screen-shot below shows.

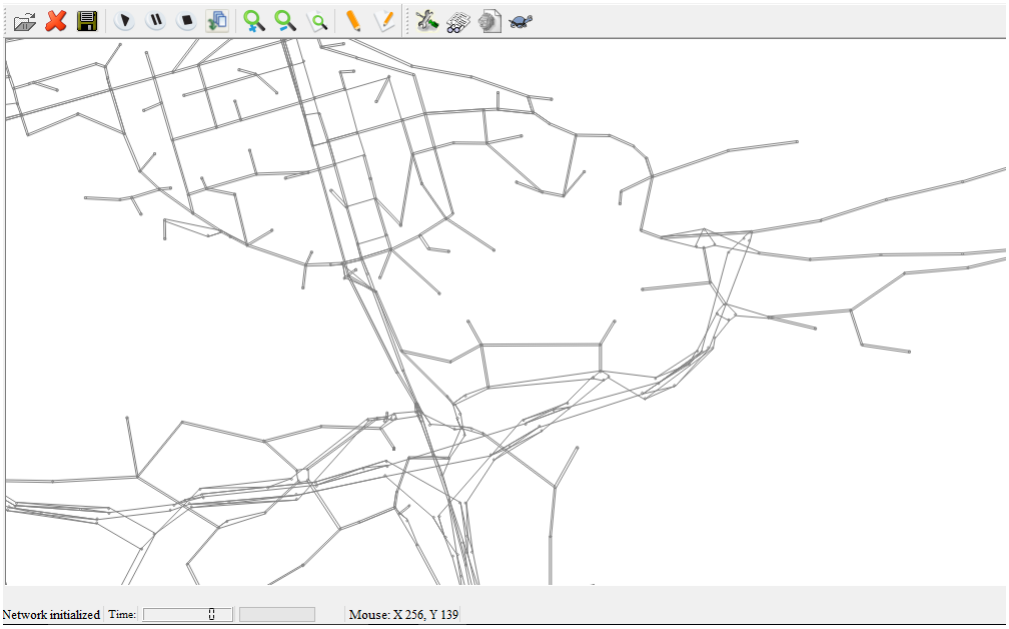

Figure 6: Road visualisation from Mezzo [16]

PTV Vissim [22] is one the state-of-the-art traffic simulators available for companies to use. Unlike [16], it can generate the roads automatically from an OpenStreetMap (OSM) file.

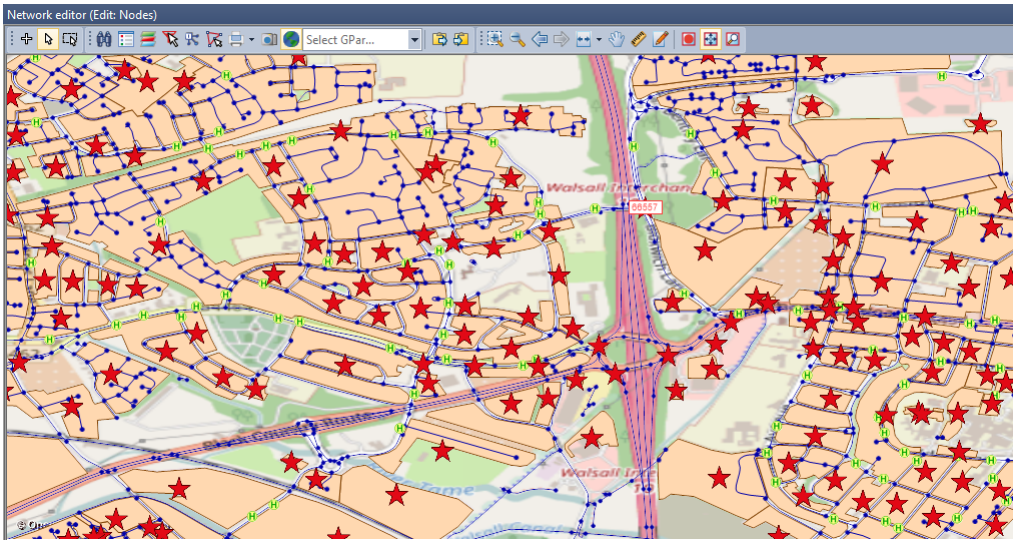

Figure 7: Road network in Vissum [22]

From the the our experience, however, the GUI is very complicated and there is an 1000 page manual dedicated to operating it. With this in mind, we can generate our main use case:

1. A user wants to simulate traffic for a specific area of map

For this project, the user should not need expert knowledge to operate the simulation. Although simple in its use, it should have similar functionality to Vissim [22], where the road network is generated automatically. The user should be able to know at any point how the simulation is progressing and when it starts and ends. The map data should not be hard to access and should therefore be stored for high availability. A high-level use case diagram can outline some of the basic interaction between the potential actors of the system:

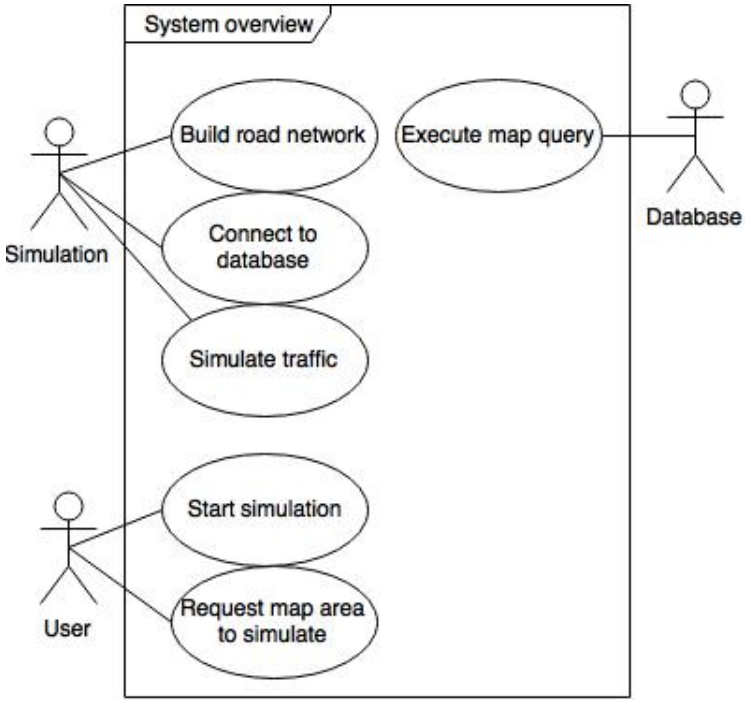

Figure 8: High-level use case diagram

3.3 Requirements Definition

Although the use case diagram shows minimal user interaction, it was important that user requirements were detailed to give the best experience; one of our objectives coming into this project was that no special knowledge would be needed to run the system. That being said, the main legwork of the project came from simulation itself. The literature highlighted three potential models that we could develop; we had to list our high level functional and non-functional requirements that encapsulated the techniques we collated from our research.

3.3.1 Functional Requirements

1. The system must incorporate real world map data

   1.1 The system must retrieve speed limit information

   1.2 The system must retrieve lane information

   1.3 The system must retrieve one-way information

For the simulation to be as realistic as possible, real road data will be incorporated. It will need to be up-to-date and be detailed enough to influence the simulation e.g. if the road is one-way or not.

2. The system must visualise the simulation in action

   2.1 The system must visualise the vehicles in some way

   2.2 The system must visualise the road network

The user needs to understand how the simulation is currently doing without having to worry about the technicalities. Both the road and vehicles need to be represented to give a more interesting experience.

3. The system must allow the user to request an area of map to simulate

3.1 The system must allow the user to precisely choose a map area

This is the main interaction between the user and the system. It allows the user to have flexibility in where they simulate traffic and how big the area will be.

4. The system must allow for different simulation inputs

4.1 The system must be able to simulate different number of vehicles

Changing the inputs given to the simulation will mean that different conclusions can be drawn. For example, how the number of vehicles effects the overall time of the simulation.

5. The system could represent cars as individuals in the system

5.1 The system must track the position of the vehicle

5.2 The system could visualise vehicles individually

The more interesting simulations, our perspective, have the ability to treat vehicles as separate entities in the simulation. More information can be queried in the simulation, such as: speeds, journey time and location.

6. The system must allow cars to follow a route

6.1 The system could allow cars to switch route during journey

Most of the simulators in the literature have the ability to send vehicles down specific routes. This allows for a more realistic network of cars, as each will have their own origin and destination; mimicking how real traffic functions.

3.3.2 Non-functional Requirements

1. Portability

1.1 The system must run on all major operating systems

2. Performance

2.1 The system must be responsive during simulation

2.1.1 The system must allow the user to change the map focus and position accordingly

3. Scalability

3.1 The system must be able to simulate a large number of vehicles

4. Delivery

4.1 The system must be implemented by the dissertation deadline

5. Usability

5.1 The system must be easy to operate for the average user

5.2 The setup time needed to run a simulation should be as short as possible

6. Availability

6.1 The system must be available to use at any time given the correct dependencies are installed

3.4 Requirements Analysis

Requirement	Microscopic	Macroscopic	Mesoscopic
1	Meets this requirement	Meets this requirement	Meets this requirement
2	Meets this requirement	Meets this requirement	Meets this requirement
3	Meets this requirement. However, the size of the area requested will be limited as microscopic simulations operate on more urban roads.	Meets this requirement	Meets this requirement
4	Meets this requirement	Meets this requirement	Meets this requirement
5	Meets this requirement	Partially meets this requirement. In [12], the notion of a 'carticle' is introduced for representing individual cars. These act more of a visualisation tool, rather than a true individual in the simulation.	Meets this requirement
6	Meets this requirement	Partially meets this requirement. Simulators such as METANET [18] have route guidance where driver groups can be controlled and sent to certain areas; it doesn't have the ability to send individuals down paths. The aggregate nature of macroscopic means individual representation is uncommon; the en-route switching requirement, therefore, cannot be met.	Meets this requirement

Table 1: Functional requirements analysis

Requirement	Microscopic	Macroscopic	Mesoscopic
1	Meets this requirement	Meets this requirement	Meets this requirement
2	This depends on the number of cars selected to simulate. Microscopic simulations are constrained by the number of individuals in the system. If this model was chosen then the number of vehicles would be restricted, depending on the processing power of the machine the simulation is being run on.	Meets this requirement	Meets this requirement
3	Meets this requirement	Meets this requirement	Meets this requirement
4	After reviewing the literature, it is cleat that it would be unlikely to get a fully-functioning microscopic simulator working in the time given for development. The reason for this is that a lot of parameter tuning is required; each individual has their own set of rules governing decision making. The calibration itself could be a project on its own.	Meets this requirement	Meets this requirement
5	Meets this requirement	Meets this requirement	Meets this requirement
6	Meets this requirement	Meets this requirement	Meets this requirement

Table 2: Non-functional requirements analysis

3.5 Chosen model

From the requirements comparison, the mesoscopic simulation was the most suited overall to the requirements specified. Given the time constraints, creating our own simulation techniques would be unwise. In order to meet the requirements, the Mezzo architecture will be used as the main focal point. To make it clear, we will adopt the link/node model, the method in which the vehicles traverse the road (using their earliest exit times) and some of the shockwave techniques described. The main challenge of the project is designing and implementing components around these so they not only work as indented in the literature, but also complement the additional requirements we specified above. The main difference will be the simulation step update; we will use a discrete

time based update, rather than an event based update. The reason for this is so that there is the possibility to get a smoother visualisation if the vehicles are displayed as individuals. Another difference is the that Mezzo does not generate the road network automatically, and does not use a map service for its data.

4 Design

4.1 Software and libraries

4.1.1 Programming Language

To meet our delivery requirements, Java was chosen as it was our most proficient language; the choice sped up development and resulted in cleaner code. The performance of Java is also ideal, although the memory management is not handled by the programmer, the JVM offers adequate performance. It also has good mutli-threading capabilities needed to parallelise the system if required. It is also object-orientated, which suits a modular approach to the architecture. Java also has good library options and is well utilised in the open-source community. It is platform independent which means it can run on all the major operating systems.

4.1.1.1 Streams

Version 8 of Java was used because it has a feature called Streams [27]. These allow for a more functional style of programming, with the ability to use predicates and functions such as map, reduce and filter on data structures. The Map function allows us to apply a function to each member of the structure. Reduce can turn the structure into one result based on the function given; one example would be summing the elements of a integer list. Filter applies a predicate to each member of the list and only those that satisfy this are processed further. This lent itself well to the chosen simulation model as the roads use the queue structure; streams allowed clean and easy-to-test solutions for various algorithms needed to query the simulation. Another advantage was speed of development; the functional style uses less code and is quicker to implement:

```
public double queueLength(double _t) {
    return this.stream()
            .filter(v -> v.getEarliestExitTime() < _t)
            .mapToDouble(v->v.getLength())
            .sum();
}
```

Figure 9: Calculating queue length using functional style

```
public double queueLength(double _t) {
    double sum = 0;
    for(Vehicle v : this){
        double _eet = v.getEarliestExitTime();
        if(_eet<_t)
            sum+=_eet;
    }
    return sum;
}
```

Figure 10: Calculating queue length using imperative style

The two code snippets were based on a queue operation. The functional style was much cleaner and the ability to chain functions was helpful for larger functions.

4.1.2 Map Data Source

The chosen data source for the road information was OpenStreetMap (OSM). The main reason for this was because it is open-source and is not owned by any company (Google Maps) so we avoided any licensing issues. It is also well documented and a number of existing simulations (such as Vissum [22] use it as an import option. It contains the ideal amount of information needed: speed-limits and lane information are included; as well as road type and if it is one-way or not. There were a few problems that arose with inconsistencies in the data; these will be explained later in 7. Another important feature is that it is regularly updated and downloads are available for certain areas (UK, Birmingham etc.); we downloaded ours from [25].

4.1.3 Map Data Store

PostgreSQL was the database system used to store the map information. It is object-relational, open source and supports PostGIS, a spacial database extender for geographical tools; this was useful when we had to store geographical points of the road. PostgreSQL can easily connect to Java using the Java Database Connectivity (JDBC) API. We had prior knowledge of this interface, so the setup time was kept to a minimum.

4.1.4 GUI library

JavaFX 8 comes with version 8 of Java and replaced the Swing tool kit as the the standard library. It was an ideal choice because it was easy to get an application running and it has a range of layouts that can be used. Another reason is that is comes with XYChart, a graphing library, so any visualisation of the output was easily integrated.

4.1.5 Map Viewer

For displaying the road network and the vehicles on it, a map viewer needed to be chosen. JMapViewer is a Java developed map tool that uses OSM for its visualisation and geographical information. This made it ideal to integrate with the system, as any mapping onto the viewer from the simulation was accurate. For example, any coordinates we retrieved from the database for a certain road, were drawn correctly onto the viewer using its MapPolygon object.

4.1.6 Dependency Management

Dependency management was used to handle external resources and libraries that needed to be incorporated into the system. Gradle is developed in Java and it allows the system to be built from a Gradle file. In this file, the dependencies were specified, with their source and version; they were able to be imported into the code if the build was successful. Gradle also allows you to organise your project structure, with the option to specify main, resources and test directories; this kept our directory structure ordered and neat.

4.1.7 Testing

Unit testing was used to test independent components of the system. Java has unit testing built in with JUnit tests; it allows conditions to be tested using *assert()* commands. Java Hamcrest [24] is an extension for unit tests that provide matchers for expressing intent in the tests. The

main reason for using this is that it has good matchers for containers, something that JUnit lacks; the only method that comes close in JUnit is *assertArrayEquals*. The ability to test containers was especially useful when testing queue operations e.g. how many vehicles are queued at a given time. The syntax looked like *assertThat(basket, contains(apple, orange, pear))*. The order of statements made it ideal for more complex tests, as the intent of the expression was much clearer.

4.1.8 Routing

In order to use the OSM data, it had to be be transformed to a route-able form where by edges and nodes are generated. This made it easier to generate our own simulation graph for a specific area. Doing this transformation by hand would have been very time consuming so we used a tool called OSM4Routing [23]. Given an OSM file for a particular area, it formed the nodes and edges; we then input these into two separate tables of a PostgreSQL database.

4.2 Architecture

4.2.1 Class Diagram

The components of the system were now known and the chosen technologies were in place. This meant that a class diagram of the system could be generated. From the literature, we knew the main components of the simulation and how they interacted. However, we had to design additional components and ensure the relations between them allow the simulation techniques to work correctly. We also had to choose the best technologies and data structures in Java. Below is a class diagram describing the classes in the system and the relationship between them. It is assumed that if there is a one-to-one mapping, there is the respective attribute in each connecting class. Also, any attributes listed in the diagram have getter and setter methods if their modifier is private.

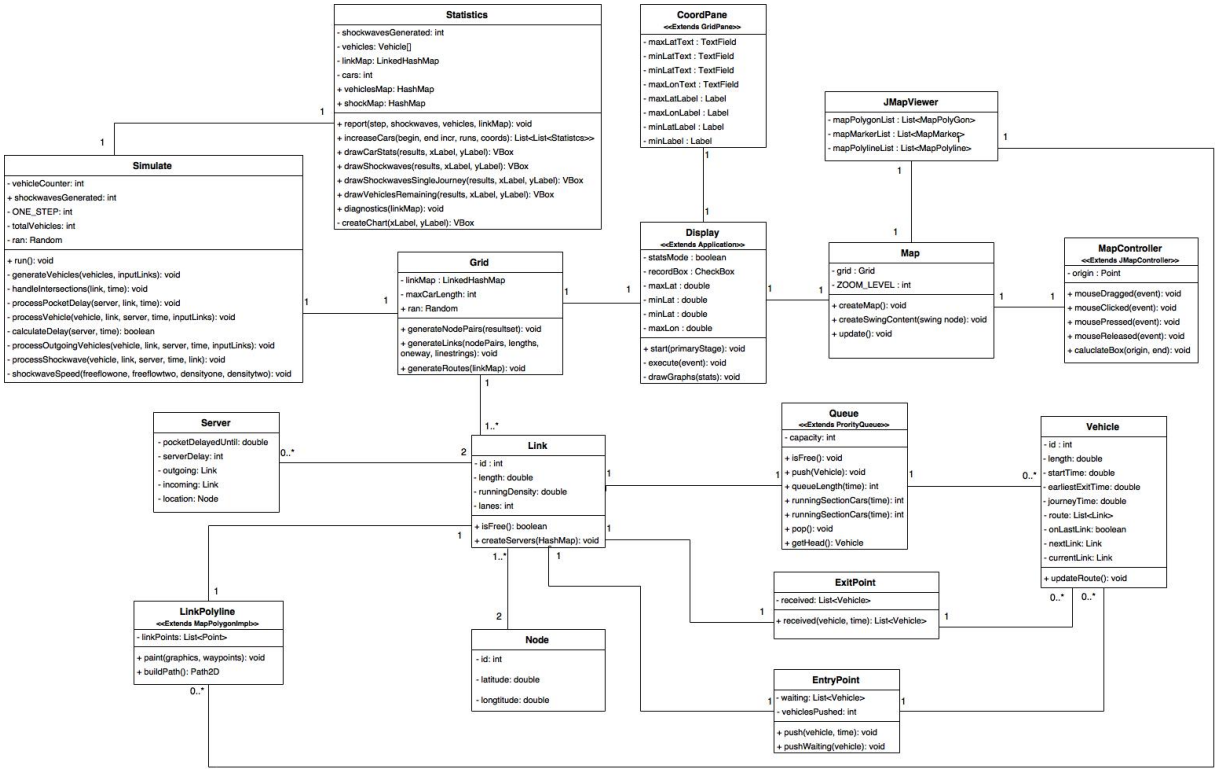

Figure 11: Sequence diagram of *handleIntersections*

4.2.1.1 Link, Queue and Node

The Link class represents the roads in the simulation and contains the attributes that describe their topology in the world, such as number of lanes. The most important attribute is *runningDensity* which describes the density on the running part of the queue. The Queue class maps to the link class, and is the data structure used to hold vehicles. We decided to separate the two classes because they represent two different parts of the simulation; the Link acts as a branch in the graph that connects to other links, whereas the queue is the underlying container that governs how the vehicles are stored on the graph, and how they move from link to link. It gave flexibility when developing the queue operations; any new implementations of the structure could be swapped in and out by implementing a QueueTemplate. The diagram shows that the Queue inherits from Java's *PriorityQueue*; this is one example of the queue implementation. A ProrityQueue makes sense because the vehicles are organised on the queue using their *earliestExitTime* i.e. the earliest time it takes to reach the downstream node of the link. The Node class is minimal and consists of a longitude, latitude and an id.

4.2.1.2 Server

The server class contains the attributes needed to process a turn in the simulation. It is its own class because it connects two links together; a separate server is needed for each outgoing turn for a specific link. The *pocketDelayedUntil* attribute is needed to simulate delays for turns; if the outgoing link is blocked then a delay is enforced until there is space on that link; this delay was governed by the speed of the shockwave produced. A link may have zero-to-many servers, depending of the connectivity of the road i.e. if it is a dead end.

4.2.1.3 Vehicle

The Vehicle class represents a separate individual in the simulation. The most important attribute is the *earliestExitTime*. This decides where in its respective queue it gets positioned and gives an indicator (delay could change it) on how long it will remain on its current link. Routes are represented by a list of Links; there is a method that updates the current and next links when the vehicle progresses. Once the vehicle has completed its route then it will get removed from the simulation.

4.2.1.4 Entry and Exit points

These were vaguely mentioned in the literature for Mezzo [16] but it was not clear how they were used. We decided to treat them as boundaries to the simulation and each Link has one of each. This let us keep track of when and where the vehicles entered and left the simulation. This was important for both ending the simulation and generating statistics needed for the evaluation. The EntryPoint class stores waiting vehicles that cannot enter the simulation because the Link is full, whereas the ExitClass stores the received vehicles that have finished their journey.

4.2.1.5 Grid

This class stores a LinkedHashMap where the key is an integer and the value is a link. A HashMap gives a constant lookup time and its entry set can also be streamed if needed. This map is used throughout the simulation as the road network graph. The class builds the links through various methods; the most important one processes a results set from a database query; here the road information (lanes information) is collated and passed to the *generateLinks()* method.

4.2.1.6 Simulate and Statistics

The Simulate class contains the methods that implement the main simulation techniques mentioned in the literature. The *run()* method contained the simulation loop and called these methods in the correct order. Each instance of Simulate has a Statistics object. This gets created when the simulation finishes and allowed us to generate graphs and spot patterns in the data. The one-to-one mapping also gave the opportunity to run the simulation in batch and to create averages of the data collected.

4.2.1.7 Map classes and LinkPolyline

The Map class embedded the map MapViewer. The viewer contains the underlying geographical information, which can hold different map objects, such as polylines. Polylines are geographical points where continuous lines are drawn in between them. This is how the view will update; each Link has a LinkPolyline describing the road and these will get added to the viewer's polyline list so they can be drawn (and updated) when necessary. The MapController class handled user interaction with the map. It has methods that fire when mouse events are triggered, such as clicking and dragging. Having a controller allowed us to implement custom event handling; this was needed if the user wanted to select a map area to simulate.

4.2.1.8 GUI

The Display and CoordPane make up the graphical interface where the map will be shown. The Display class was the point of execution and had buttons for the user to start the simulation. There is a boolean *statsMode* to determine whether statistics should be collated; this is ideal for batch runs.

4.2.1.9 Utility classes

There were utility classes that contained static functions that were used frequently and did not need an instance to be run. They were not included in the class diagram because they did not form any relation. The classes are as follows:

1. QUtil - this contains the functions for queue operations. Such examples include: determine which vehicles are queued at a given time and the distance in front of a certain vehicle.
2. MapUtil - this contains functions for map operations. This included: calculating source and target nodes in a connected graph, and functions for comparing latitudes and longitudes.
3. GridUtil - this class contains functions used to query the grid. It has operations like calculating the number of vehicles entering and leaving the grid.

In addition to these, there was a separate class Query, used for handling the connection to the database, executing queries and retrieving the result sets.

4.2.2 Activity Diagram

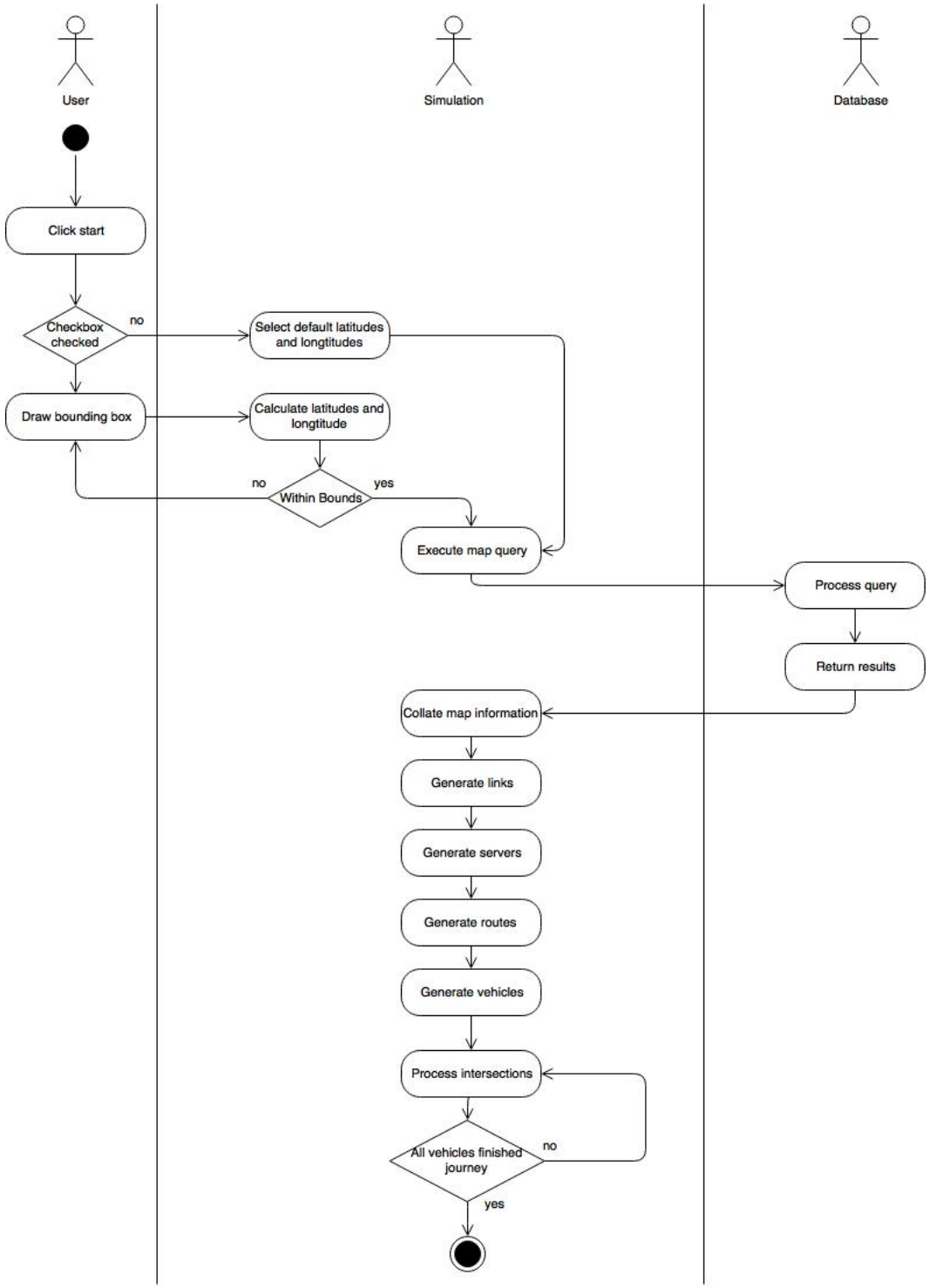

Figure 12: Activity Diagram of a full simulation

The activity diagram above shows the flow of of one activity to another. The majority of the activities were within the simulation itself and were concerned with generating the road network.

The handling of the intersections is where the main logic took place and it is detailed in the sequence diagram below.

4.2.3 Sequence Diagram

The *handleIntersections()* method is detailed below. The sequence diagram lets us capture the flow of messages between the objects, which are arranged in a time orientated manner. The method is split into three parts: processing a delay, calculating a delay, and moving vehicle onto their next link. We used our research and design decisions to generate the specific flow. In the diagram, the delay calculation is not detailed due to it's size and complexity; instead, it is explained in 5.5.

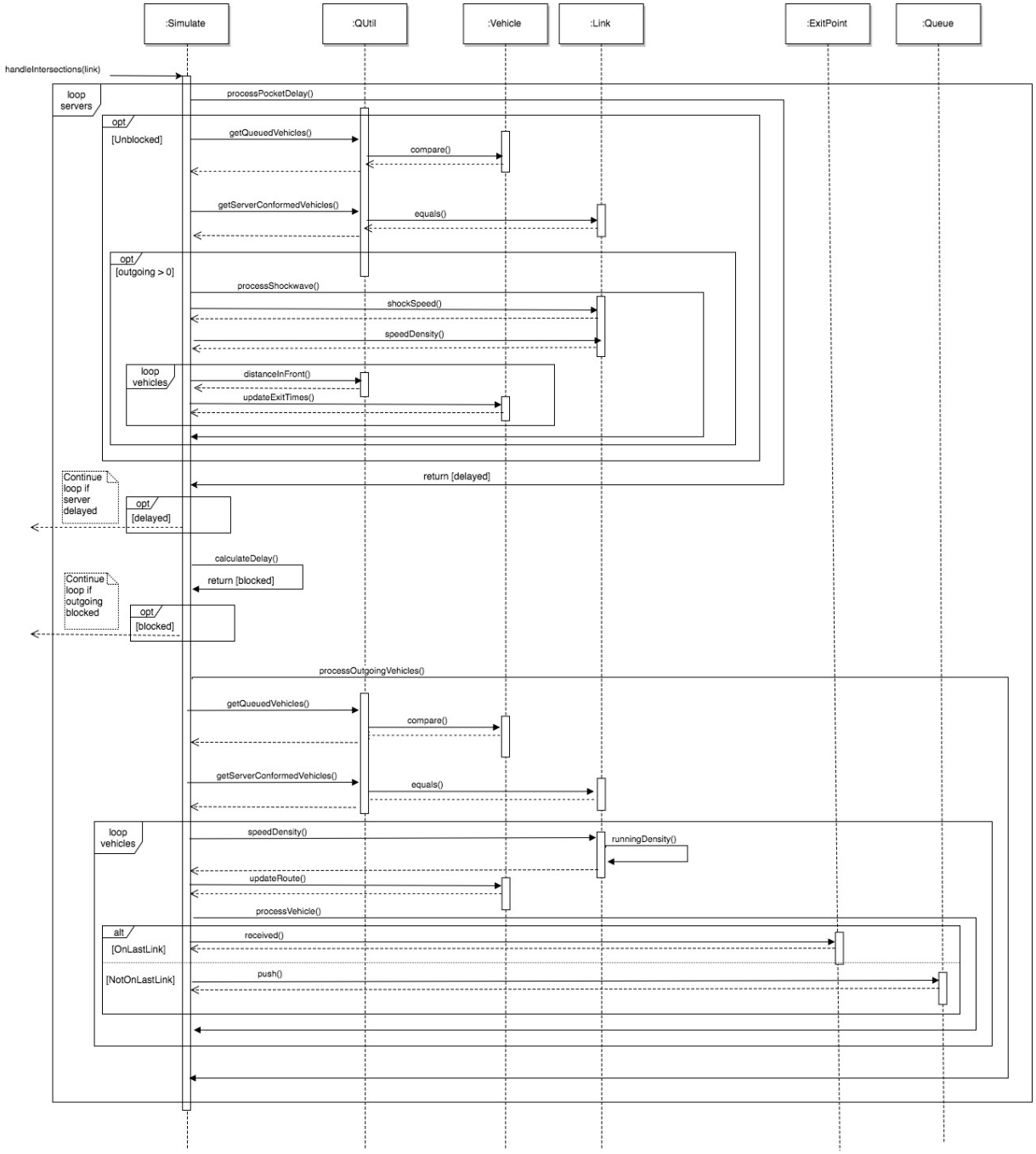

Figure 13: Sequence diagram of *handleIntersections*

5 Implementation and testing

5.1 Software Methodology

We took a bottom-up approach to development because we had knowledge of some of the modules from the literature. Each component was developed and tested independently before moving onto the next. This ensured that any functionality that could be tested on its own worked before integration. Although no strict methodology was used, it was loosely based on the Agile and Extreme Programming techniques; each component was seen as a separate iteration, where constant development, testing and refinement were carried out. Once the component was complete and had met some form of testing criteria, we moved onto the next one, integrating where necessary.

Below, the components and their implementations are described.

5.2 Queue operations

Our first task was to create the queue operations needed in the simulation; these were formed by analysing the literature in 2.2.4. These functions are called regularly and query the queue and the vehicles on it.

```
public int runningSectionCars(double _t) {
    return this.stream()
            .filter(v -> v.getEarliestExitTime() >= _t)
            .collect(Collectors.toList())
            .size();
}
```

Figure 14: Calculating queued cars at time $\_t$

Using streams, we filter the cars that are in free flow using their earliest exit time and checking if it is less than the current time, $\_t$. We then return the number of the cars that meet this predicate.

The above structure was used for most of the queue functions. For each operation we can generate a unit test that checks if the operation meets all the edge cases. For testing the queue structure, the cases ensured that empty queues were dealt with when necessary; some functions would never be called unless a queue was occupied. Using the Hamcrest extension explained in 4.1.7, the function below demonstrates a test for calculating the distance in front of a queued vehicle:

Given four test cars and their respective exit times, tests are constructed to ensure that that the calculation is true. The literature was not clear on how the distance in front of a vehicle was worked out; we take a sum of the lengths of the cars in front of the one we are comparing. One of the more important edge cases above is checking that when calculating the distance where two cars have the same exit time, the one positionally ahead in the actual queue structure is used, rather than returning zero. Although this may seem like a simple test, the simulation could not function without the test passing. Similar tests to the one above were constructed for different queue operations. Building up this test suit was useful because if any changes in other modules were developed later on, we could see if the tests would still succeed.

```
@Test
public void testDistanceInFront(){
    Queue queue = new Queue();
    Vehicle v1 = new Vehicle(1);
    v1.setLength(4);
    Vehicle v2 = new Vehicle(2);
    v2.setLength(4);
    Vehicle v3 = new Vehicle(3);
    v3.setLength(4);
    Vehicle v4 = new Vehicle(4);
    v4.setLength(4);

    v4.setEarliestExitTime(1);
    v2.setEarliestExitTime(2);
    v1.setEarliestExitTime(3);
    v3.setEarliestExitTime(4);

    queue.push(v1);
    queue.push(v2);
    queue.push(v3);
    queue.push(v4);

    double inFrontOfFirst = QUtil.distanceInFront(queue, v4);
    assertThat(inFrontOfFirst, is(0.0));

    double inFrontOfSecond = QUtil.distanceInFront(queue, v2);
    assertThat(inFrontOfSecond, is(v4.getLength()));

    double inFrontOfThird = QUtil.distanceInFront(queue, v1);
    assertThat(inFrontOfThird, is(v4.getLength() + v2.getLength()));

    double inFrontOfLast = QUtil.distanceInFront(queue, v3);
    assertThat(inFrontOfLast, is(v4.getLength() + v2.getLength() +
       v1.getLength()));

    //Same _eet
    v2.setEarliestExitTime(1);
    double same = QUtil.distanceInFront(queue, v2);
    assertThat(same, is(v4.getLength()));

    v3.setEarliestExitTime(1);
    same = QUtil.distanceInFront(queue, v3);
    assertThat(same, is(v4.getLength() + v2.getLength()));
}
```

Figure 15: Unit test for calculating distance in front of a queued vehicle

5.3 Vehicle traversal

In our simulation, the vehicles don't explicitly 'move'; whenever a vehicle enters a link the speed-density function is used to give the speed they'll travel along the link. For the speed-density function to work, the running density along a link is calculated for a given time. This uses the function described in 5.2 to retrieve the number of cars on that section. This is then divided by the length of the running section. It was not made clear how the running section length was calculated; we could either sum the car lengths in the running section, or sum the cars lengths in the queue section and subtract this by the link length; the two methods yield different results. If the latter method is used, the running section equals the link length if there a no vehicles are present; this is incorrect as no cars are in free flow. We decided to use this method but enforced an additional check if no vehicles are present - the running length would be zero if the case.

Once we had the running density, it was used to give us the speed of the vehicles by applying it to the speed-density function. The only thing left to do was calculate and assign the earliest exit time to the vehicle; this was achieved by dividing the link length by the speed acquired. Once their exit time is less than the current time, they can be queued and moved onto the next link

5.4 Intersections

Switching from one link to another was handled by queue servers; every time a vehicle is ready to switch link a check is made to see if the server we're currently looking at matches the vehicle's next destination. If so, it can be processed and pushed onto the new queue. More than one vehicle can be processed per call of the *handleIntersection()* method. We made the choice to iterate through the servers of each link, rather than iterate through the vehicles, as the servers act as a higher level mechanism through which vehicles can be grouped. The snippet below is from the *handleIntersctions()* method and shows how one link' servers are dealt with:

```
for (Server server : link.getServers()) {
    boolean delayed = processPocketDelay(server, link, time);
    if (delayed) continue;

    boolean outgoingBlocked = calculateDelay(server, time);
    if (outgoingBlocked) continue;

    // Outgoing link free
    processOutgoingVehicles(link, server, time);
}
```

Figure 16: Processing a turning for a link

There are two booleans that need to be satisfied before a vehicle can be moved on; they ensure that there are no blocks and delays and are described further in 5.5.

5.5 Shockwaves

Implementing shockwaves was quite challenging but one of the more interesting parts of the simulation; the function used to calculate the shockwave speed is shown below and was translated from the literature.

```
public double shockwaveSpeed(Link upstream, Link downstream, double
    time){
     double uK = upstream.getkMax();
     double dK = (((double)downstream.getQueue().size()) /(
        downstream.getLanes()*downstream.getLength()));
     double dV = downstream.speedDensity(time);
     double dQ = dK * dV;
     double ss =  dQ / uK - dK;
     return ss;
}
```

Figure 17: Calculating the speed of a backwards recovery shockwave

We retrieved the densities for the waiting upstream link with the lower flow and higher density, and the downstream link that has lower density but higher flow of vehicles. Note that the density of the downstream link, dK, is taken across the whole link rather than the running part, as this can be empty and will give zero shockwave speeds; we had to take the density of the queued vehicles too. dQ represents the flow of the downstream link, it is divided by the difference in densities of the two links to give the speed of the shockwave.

The integration of this function was challenging due to the number of steps involved. The literature was ambiguous in relation to certain situations. One of the main problems that arose was how to process shockwaves for links with more than one server. The diagram below shows the situation:

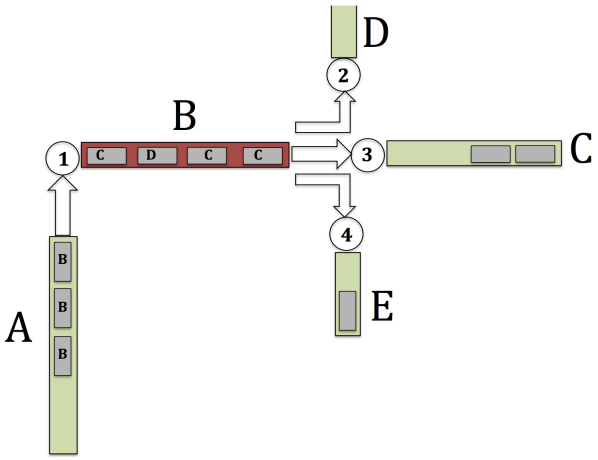

Figure 18: Calculating shockwave

The diagram shows vehicles, links and their servers; the letters on the vehicles correspond to the next link in their route. Link A is waiting for link B to become unblocked. The speed of the shockwave that will move backwards across B is calculated from the difference in densities of itself and its outgoing links (C,D,E). We were unsure if we took the densities from the outgoing link that the first queued vehicle on B is heading towards (link C), or whether we take an average of all three. As the above shockwave function shows, the speed of the shockwave is determined by the difference in two traffic conditions; if we chose the outgoing link that had higher density (link C), then the speed of the shockwave would be slower and the vehicles on link A would be waiting longer. Due to this, we decided to take an average of the densities and flows of the

outgoing links and avoid any outgoing links that none of the vehicles (on link B) are heading towards; link D would be avoided in this case. Once the delay for link A is calculated, it will get assigned to server 1. The psuedocode for processing a blocked link, as per figure 18, is outlined below:

Algorithm 1 Calculating a delay

```
1:  procedure CALCULATEDELAY(SERVER, TIME)
2:      isFree ← checkFree(server)
3:      if !isFree then
4:          shockwavesGenerated++
5:          outgoingLink ← getOutgoing(server)
6:          outgoingServers ← getServers(outgoingLink)
7:          if size(outgoingServers) > 1 then
8:              occupiedLinks ← getOccupied(outgoingServers)
9:              if size(occupiedLinks) ≠ 0 then
10:                 shockSpeedSum ← streamAndSumShockSpeed(occupiedLinks, time)
11:                 shockSpeed ← shockSpeedSum/size(occupiedLinks)
12:         else if size(outgoingServers) > 0 && isOccupied(first(outgoingServers)) > 0 then
13:             shockSpeed ← shockSpeed(outgoingLink, first(outgoingServers), time)
14:         else
15:             return false
16:         delayedUntil ← length(outgoingLink) / shockSpeed
17:         getPocketDelay(server) ← delayedUntil
18:         return true
19:     return false
```

5.6 Simulation Loop

Putting all the components together was very tricky due to the large amount of moving parts. We knew that each link needed to be processed individually so it made sense to implement a simulation loop. In the loop the top level function would get passed a time relating to the current loop counter; this would work its way down to the deeper levels of the call-stack. This ensured that each loop was working in the same time interval; no link or vehicle would be ahead or behind. As we were using a discrete-time based update, the counter was incremented by the *ONE_STEP* variable, which was set to one by default. For each loop, every vehicle in the simulation is processed; the flow and methods are outlined in the sequence diagram in 4.2.3.

The method to end the simulation involved the entry and exit points. A vehicle is pushed onto an exit point when they have finished their journey. When the total number of cars received by all exit points, sum to make the total number of vehicles we put on the grid, then we know that the simulation has finished.

A delay is present in the loop to slow down the rate at which the simulation runs. This was implemeneted by making the thread sleep for a certain period of time. It was needed so the that the updates to the

The above method collated the number of vehicles that have finished their journey. The check is made every loop and it ensures the simulation does not run forever.

```
public static int totalVehiclesOutput(HashMap<Integer, Link>
    linkMap){
     return linkMap.entrySet().stream()
             .mapToInt(l->l.getValue()
                         .getExitPoint()
                         .getReceived()
                         .size())
             .sum();
}
```

Figure 19: Calculating total vehicles that have finished their journey

5.7 Hand-built Grid

Before we could move onto integrating real map data, we needed to check that the simulation was showing expected patterns in various outputs; these are explained in 6.1.1 and 6.1.2. To get these patterns we had to construct an artificial grid with artificial links and servers. Although the grid itself was not exciting, we were able to put vehicles onto this and record the simulation output. To create the grid, we had to create node pairs and join them together based on their common nodes to form a well-connected graph.

5.8 OSM Grid

Once we had established the correct patterns in the data, incorporating OSM data was the next step. The area we imported was limited to Birmingham, UK because the size of the OSM file was relatively small and it gave a sufficient area for testing. A lot of data in the file needed to be cut down to avoid roads that we did not want the vehicles travelling down, such as service roads, footpaths and car parks. The following steps were used to achieve this:

1. Use *Osmosis* tool to import unprocessed OSM file into database X to get ways information
2. Use SQL query below to filter out roads (in database X) we need and import rows into database Y
3. Use *OSM4Routing* tool on unprocessed OSM file to generate edges / nodes tables and import into database Y
4. Use SQL query in database Y to cross-reference the nodes table with the ways table when map data is needed

The ways table generated from step 1 gave detailed information about the road topology; the tags column contains key-value pair arrays that can be queried - the key ‘highway’ is used to filter only the roads we want:

Step 2 would not have been possible if we applied the above query on the route-able data gathered in step 3 because *OSM4Routing* does not keep the ways data; we would have had no way of querying the road type of the rows in the edge table. A single row of the edges table consisted of road length, source node, target node, car accessibility, one way information and a linestring. Each node has a latitude, longitude and id. Our method allows us to keep all the connected edges even after filtering the roads in step 2. We could now cross-reference the edges / nodes table with the ways table when the user requests a piece of map data - given four

```
COPY(SELECT id, tags, nodes
        FROM ways
        WHERE tags->'highway' =
           ANY('{motorway,trunk,primary,secondary,tertiary,
                unclassified,
                resedential,motorway_link,trunk_link,
                primary_link,secondary_link,tertiary_link}'
                ::text[]))
TO '/Users/Arun/Desktop/ways2.csv' WITH CSV HEADER DELIMITER ';';
```

Figure 20: Removing unnecessary road types

latitude and longitude pairs, a query (consisting of three parts) could now be executed in our *Query* class.

The ResultSet result returned by the query can be processed and used to generate real links, servers and routes. For each edge, either one or two links are generated with their nodes; if the one way flag is set to false, then the edge information is reversed, including the linestring points. These points were turned into coordinates for the *LinkPolyline* which every Link had; every polyline object was also added to the list of polylines for the map viewer so they could be displayed to the user. Once the links were in place, they needed to be connected together using common nodes; if a target node of one link equals the source node of another (ignoring links that have been reversed) then a server connects the two together. Once this grid was in place, the vehicles and their routes could be created.

//Problems with speed-limits

5.9 Vehicle and route generation

Each simulation run has a starting number of vehicles that get put onto the grid. Every vehicle gets assigned a route and are placed onto the grid immediately if there is space on their entry point (start of their route). Initially, we wanted to use the *GraphHopper* library [26] to generate more realistic routes for us. The challenge was to make translation from the points the library gave us for a route, to our simulation links. For a period of time, it was thought that our implementation was working as intended; however, a rendering bug meant we didn't spot the incorrect translation until further on in the project. We ran out of time and could not implement the method in time; this was a shame as it meant we would have to generate the routes ourselves.

Generating and assigning the routes was the source of much variability in the simulation output and is discussed further in 6. Deciding which links to start on was not a subject of the literature; it was, however, explained that each vehicle had a origin and a destination. We made the choice of filtering the starting links based on the number of servers they had; if it was greater than 2 then we knew that it was likely that they were highly connected. To generate the route, we iterated though the servers of the current link, choosing the next link to either be a random outgoing link, or choosing the outgoing link that had the highest number of servers. This was repeated until we found a duplicate link or a dead end. Although this method may not have given the most realistic routes, it gave a good a good traversal of the grid without using a path finding algorithm.

5.10 User interface

The focal point of the graphical interface was the map of the roads and vehicles. We therefore made it cover most of the dialogue box that contained all of the components. After embedding the map viewer into JavaFX, we needed to implement the method that updated the GUI throughout the simulation. A timer was used to trigger the updates; every time it fired the map viewer's *repaint()* method was called. This forces the viewer to iterate through all the LinkPolyline objects and re-draw them using their *paint()* method. The interval for this timer is shorter than the sleep interval in the simulation loop; this ensures that at every simulation step, there will be one update to the GUI. Unfortunately, we did not have time to draw the vehicles as individuals so instead the vehicles are represented by the density of the road. If a link is occupied it is coloured red, if there are no vehicles on it then is coloured green. The screen shot below shows when there are no vehicles on the grid.

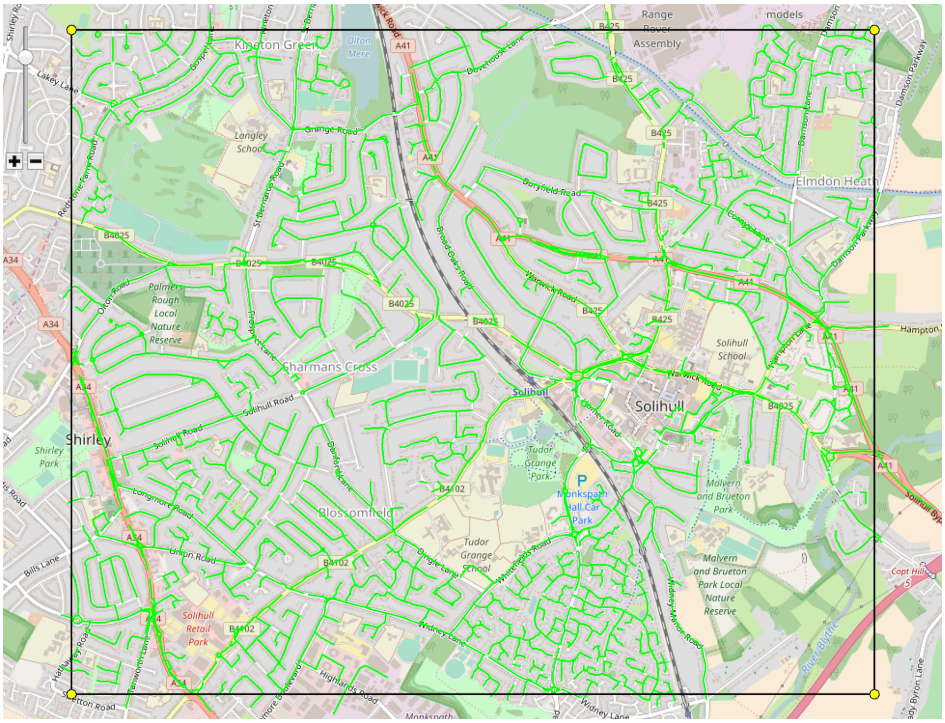

Figure 21: Empty grid

Some of the roads do not have a colouration due to a bug in JMapViewer; for some reason some polylines are skipped when repainting - we could not find the source of the issue. The black box signifies the boundary of the area we are simulating and is drawn by the user. A check-box is displayed on the GUI and if checked, the user can draw the box. To achieve the box drawing we had to change methods in the map controller that dealt with mouse movements. There were default implementations that had functionality for dragging, pressing and releasing. We modified these to add an additional check to see if the user wanted to draw the box; if so, a mouse drag would not move the map but draw the box instead. We also needed to record the origin and end points generated when the mouse is pressed and released respectively. These two points gave us enough information to calculate the four points of the bounding box and the maximum/minimum latitudes/longtitudes. The coordinates are checked to see if they are within the bounds of the database data (in our case, Birmingham). Either an error warning appears, if they are out of bounds, or they are passed to the database query for grid creation.

Below the map there is a separate pane for user input. In addition to the check-box for drawing the map area, there is a start button for beginning the simulation and another check-box for statistics mode. In statistics mode, batch runs can be made and graphs generated; to speed up the process the sleep time in the simulation loop is set to zero. The coordinates for the current simulation are also displayed; they are set to a default map area if the user doesn't wish to draw their own.

Below is a screenshot of the simulation in action with the user pane:

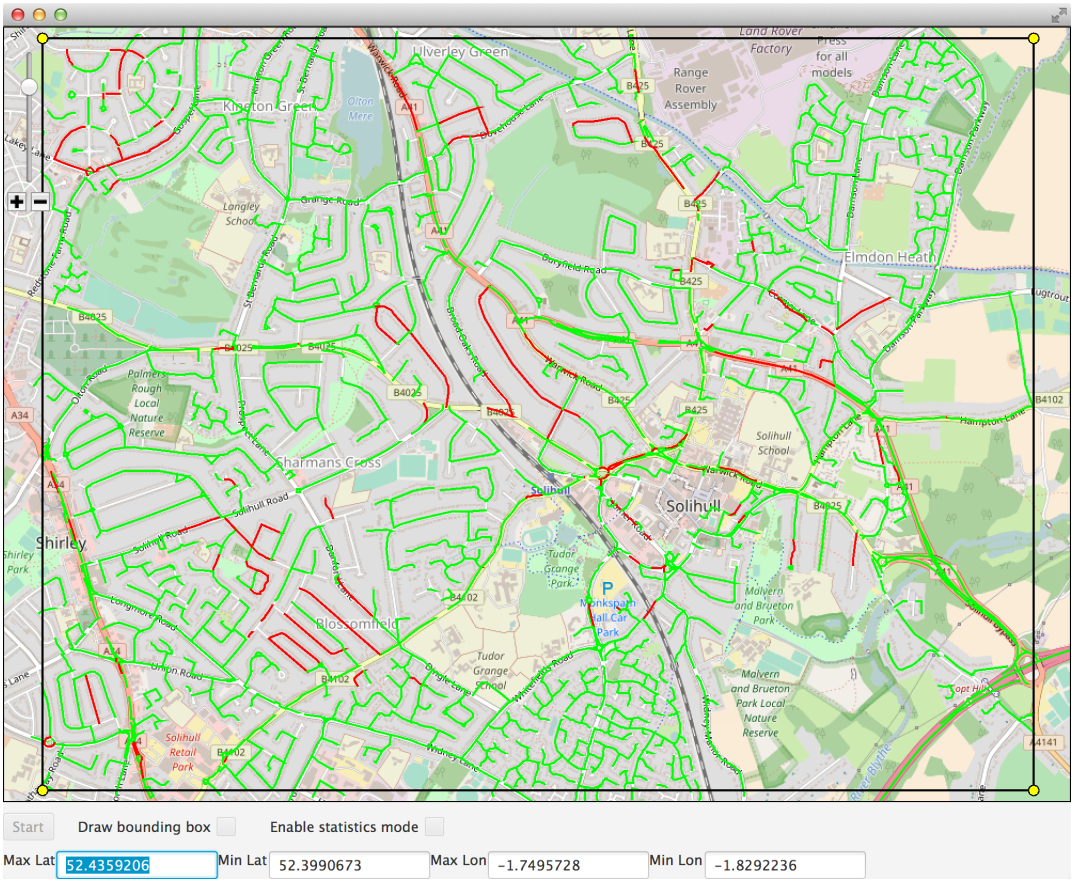

Figure 22: Empty grid

The buttons and check boxes are locked during simulation to inform the user that the simulation is still running. As the red lines show, the grid can get very busy; more so at the start of the simulation, when more vehicles are present.

5.11 Statistics

The statistics mode allowed us to generate information about a simulation run and plot the data using graphs. For each run, data is recorded and stored in relevant data structures; for example, a *HashMap* is used to map the number of shockwaves to the current simulation time. These data structures are then passed to a Statistics object and stored for analysis and graphing. The main method in this class was *increaseCars()*; it gave us the option to run a simulation several times for a specific number of vehicles and average the results. Its parameters included the start number of vehicles, end number of vehicles, vehicle increment and the number of times to run

per vehicle total. It returns a list containing the simulation runs for each vehicle increment, which allowed the mean and standard deviation to be calculated and plotted. We were able to take advantage of Java's *Future* and *Callable* objects to run each simulation (for a particular vehicle total) in parallel. For example, if we wanted to run a simulation 10 times using 100 cars and collate the results, we would not have to wait for a specific run to finish - we would start them all and join them when they're complete; this sped up the batch runs significantly.

6 Evaluation

6.1 Patterns

Given different simulation environments, we could record the behaviour of the vehicles and see what parameters effect the output the most.

6.1.1 Simulation time

The time taken for the simulation to run is dependant on the number of vehicles and their respective paths though the grid. We should be able to see an increase in simulation time as the number of vehicles increase. The results below were gathered by increasing the total vehicle count and running the simulation 50 times for each count. The mean and error bars were calculated for the data so we can make observations on the output.

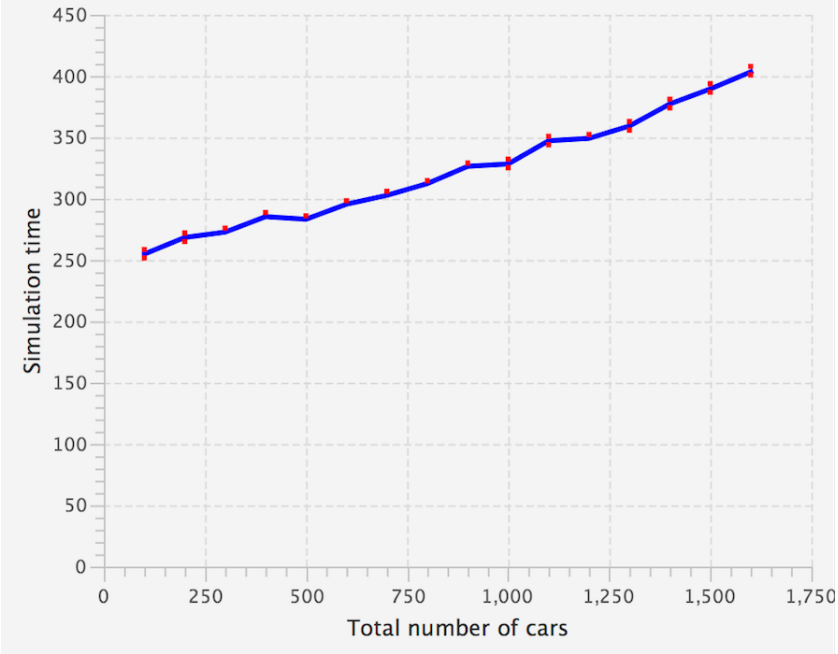

Figure 23: Simulation time for increasing vehicles with standard error

The standard error has been plotted because it allows us to compare the mean at different vehicle totals by scaling their respective variability by the number of simulation runs. The plot above shows that there is a general increase in the overall simulation time as the number of cars increase. We can safely say this because the error bars are very narrow, suggesting that the mean represents the true simulation time. There is limited overlapping of the error bars at lower number of vehicles. Between 1000 and 1300 cars, there are a few overlaps, suggesting that the simulation time does not differ as significantly for these points.

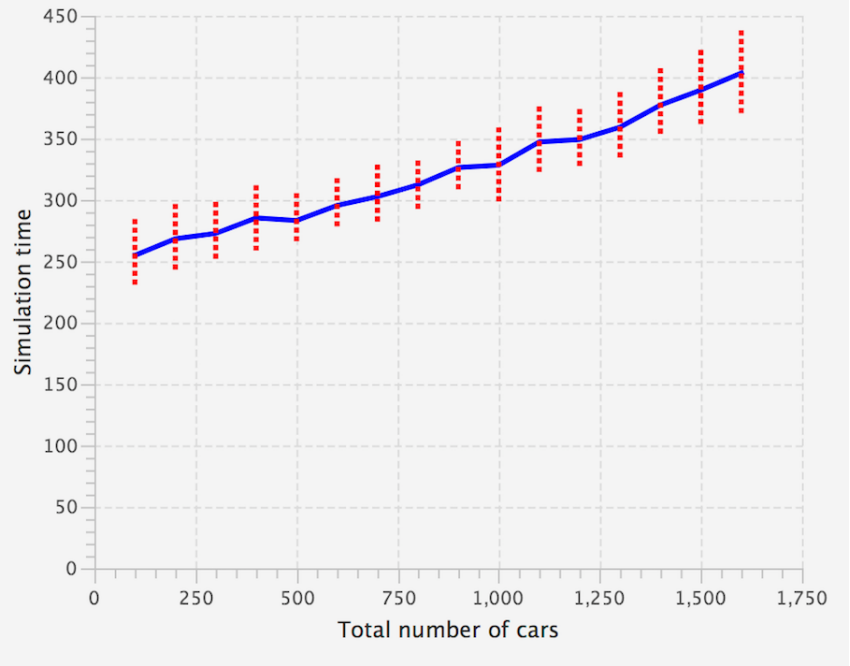

Figure 24: Simulation time for increasing vehicles with standard deviation

The dotted bars above represent the standard deviation and show that the variability of the points around their respective means is roughly the same for all vehicle numbers. The reason for this variability can be attributed to the randomness of the routes that are assigned to vehicles - although the same grid was used, none of the routes were the same. The simulation time sometimes levels out, or decreases because either the routes have an overall smaller journey time, or the routes are more sparsely generated; both of which negate the increase in cars. One way to confirm this is by setting all vehicles to have the same route.

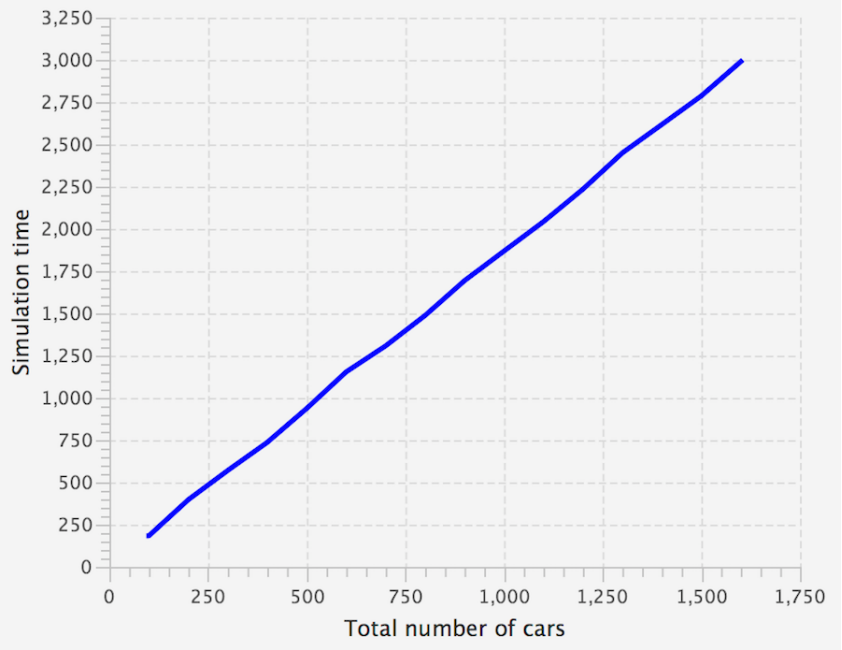

Figure 25: Simulation time for increasing vehicles with same route

The graph above shows that when all the vehicles have the same route, there is no variability in the simulation time. Even though the tests were run on the same grid above, there is now a linear relationship. This is what we would expect; more vehicles travelling down the same road would lead to higher waiting times because at a given time in the simulation, the density on the route roads is higher.

6.1.2 Shockwaves

Similarly to 6.1.1, we would expect the number of shockwaves generated to increase as the number of cars increases. Again, a higher number of vehicles results in higher density on the grid and increases the likelihood that a link will be full; this in turn increases potential delays and shockwave formation. The results below were taken from the same runs as 6.1.1.

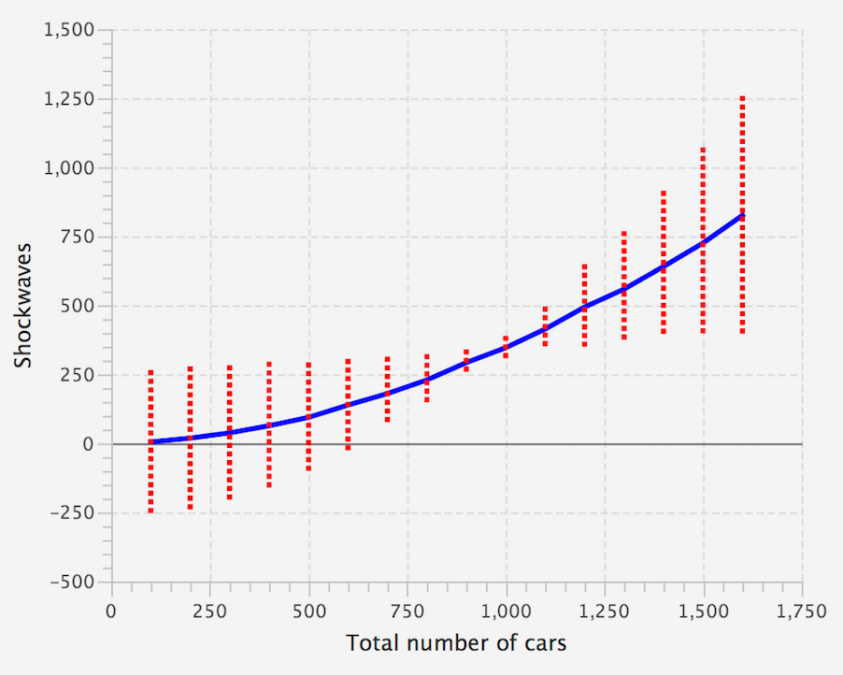

Figure 26: Shockwaves against increasing vehicles with standard deviation

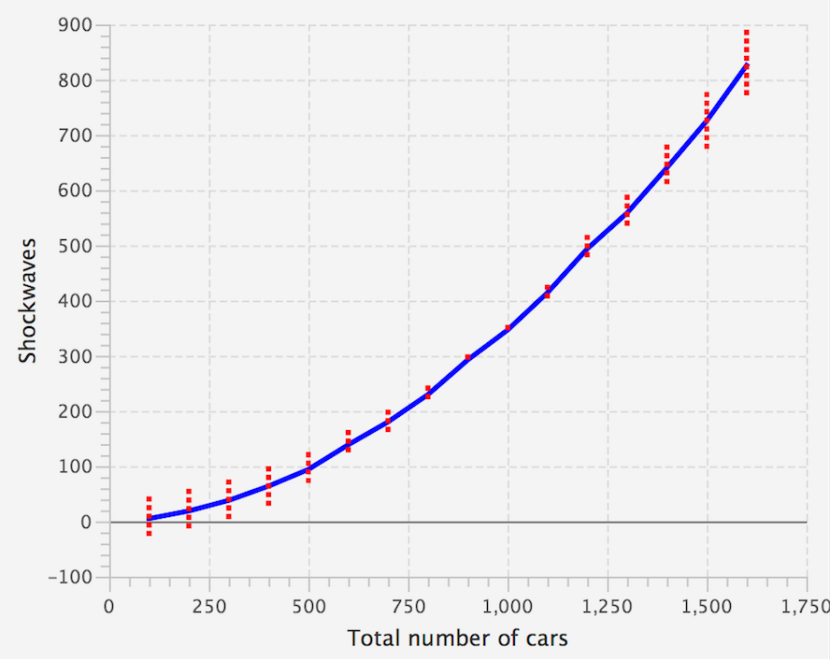

Figure 27: Shockwave generation against increasing vehicles with standard error

The graphs above shows three grouping; 0-600 (group 1), 700-1200 (group 2) and 1200-1600 (group 3). The fact that the standard error bars of the three groups don't overlap, affirms our expectation that shockwave generation increases with the number of vehicles. The standard error within the groups 1 and 3 overlap, which his signifies that there is a lower likelihood that number of shockwaves generated differ significantly from each other. Similarly, in regards

to standard deviation, there is more variability within both group 1 and 3; this means when simulating smaller and larger number of vehicles, the points differ to a greater extent from their mean. A clear pattern can be seen, in which the variability in groups 1 and 3 are respectively less than and greater than those in group 2. The reason for this pattern could be based on the grid we simulated on and its road topology. We had a suspicion that the variability of total empty space throughout the simulation had an affect on shockwave variability; empty space is defined as the total length of empty road divided by the total road length of the grid. If this proportion, averaged over a simulation run, gives us the same (or opposite) trend as above, then our assumptions would be correct. Unfortunately, after running this experiment the resulting variability remained the same.

We could not pin-point the reason for the trends in 26 and 27, but it is most likely to do with the relationship between the grid and the number of vehicles put on it. This is promising because it tells us that for a given map area, there is a certain quantity of vehicles that behaves the same; remember that shockwaves are formed because of links being full and delays being enforced. Low variability of shockwave creation suggests consistent delays for vehicles. For a real world application, the simulation could be used to predict delays and congestion; the cars that are within the 'sweet' spot (group 2) could be used as a basis when limiting and predicting the flow of vehicles, such as in: road layout changes, roadworks or toll introductions.

In addition to the experiments above, we can also show how the number of shockwaves generated throughout a single journey changes.

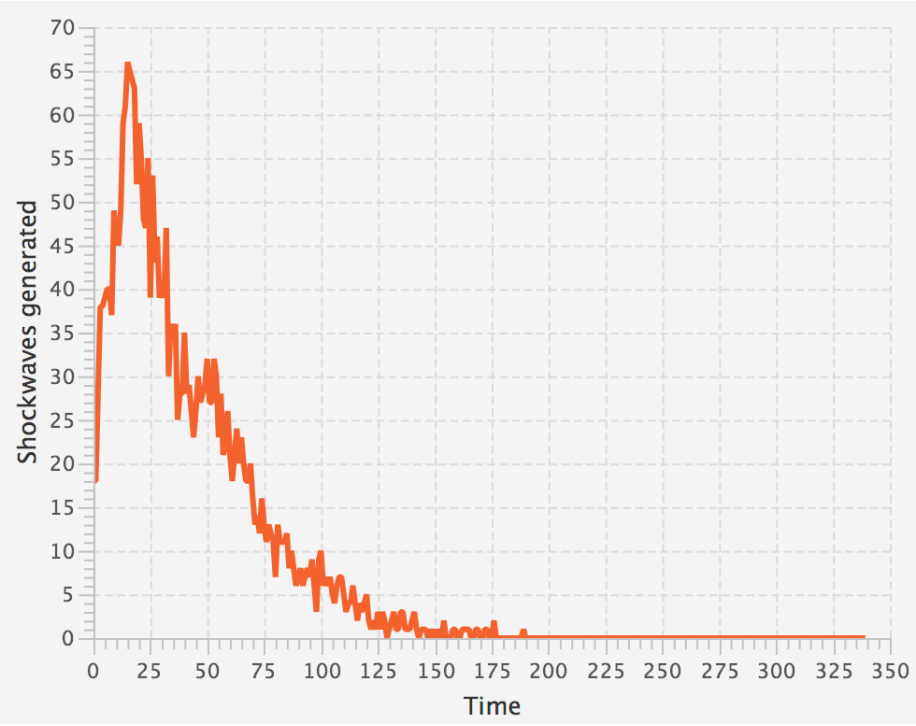

Figure 28: Shockwaves over a single journey with 10000 vehicles

The graph is not a cumulative sum of the shockwaves, it is a plot of the number of shockwaves generated per simulation step. There is a peak in the number of shockwaves generated, followed by a sharp decrease that gets more gradual as the journey progresses. This peak is expected because the number of vehicles in a single journey also rises and falls. The peak represents the

time where the number of vehicles on the grid it at its highest. As discussed in 5.9, as many vehicles as possible are pushed onto the grid at the start of the simulation; this is why the peak appears early on. More vehicles equates to higher densities on the roads, so there is an increased chance that a link is full, generating a subsequent delay and a backward recovery shockwave. As the vehicles leave the grid, the density on the roads decreases, meaning it is less likely for a link to be at maximum capacity. Due to the randomness of the routes, the number of shockwaves generated has a yo-yo pattern; if the vehicles were following the same route the line plotted would be a lot smoother.

6.1.3 Rate of exiting vehicles

Another experiment we wanted to conduct was the rate at which the grid empties with respect to the duration of the simulation. The graph below describes this relationship.

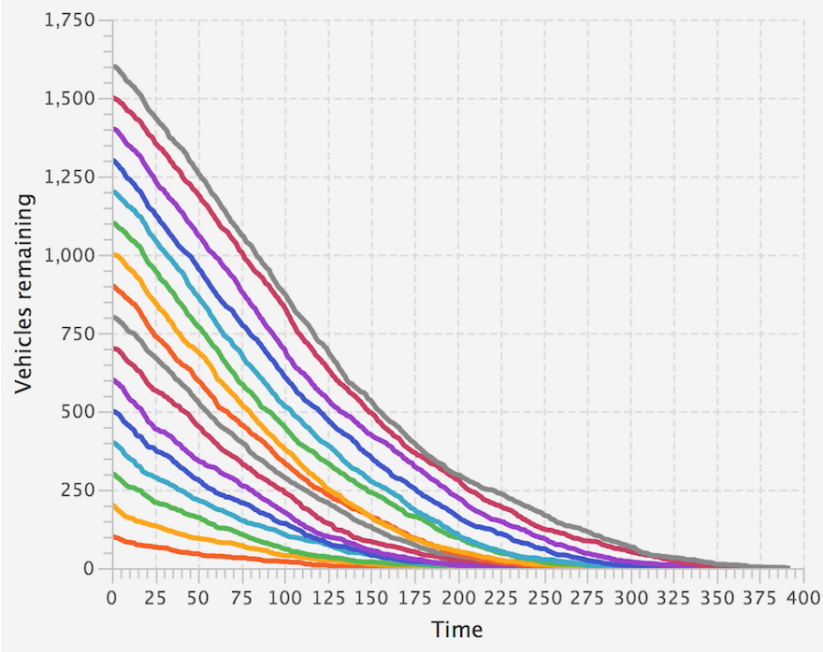

Figure 29: Rate of vehicles exiting against simulation time

The plot above does not look that promising. It tells us that at a higher number of cars, the rate of the vehicles exiting the simulation is higher than with a lower number of cars; this is unlikely down to any route randomness. The problem demonstrates that the speed-density function we used was not as effective as it should be. Although more than one vehicle can leave per simulation step, every time a group of vehicles get pushed onto the same link, the density should change per vehicle push and therefore affect (decrease) their speed. What we see here is almost a 'platooning' like effect, where vehicles are moving in groups from link to link but their speeds are not being penalised by the speed-density function - it makes it seem like the vehicles are travelling faster when grouped, then individually. A way to fix this would be to change the speed-density function so that it gives more granularity with its speed assignment. We unfortunately ran out of time before it could be experimented with.

6.1.4 Scalability

6.1.4.1 Gridlock

Without OSM integration, the simulation could scale well for large number of vehicles (> 100,000) without any problems. The OSM integration affected the scalability due to the increased connectivity of a real road graph; the hand-built graph was not nearly as complex, with a server having at most three outgoing links. The increased connectivity of the OSM grid meant that there could be many tight sections of road. With the experiments described in 6.1.1, 6.1.2 and 6.1.3, as many vehicles as possible were pushed onto the grid at the start of simulation, and at any time a space was free on an entry point, subsequent waiting vehicles were pushed onto the grid immediately. The problem with this method is that the vehicles can get stuck in one position due to these tight sections. A cyclic gridlock can form between the leading vehicles (head) of certain links, where they are all waiting for their destination link to free up. This creates a relationship between the size of the map area and the number of cars being simulated.

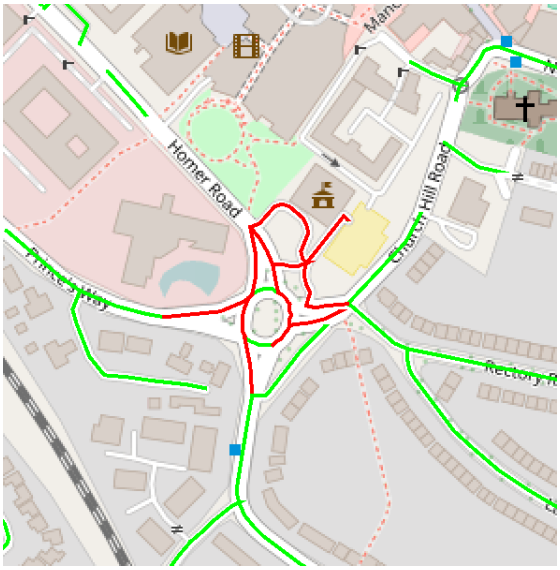

Figure 30: Gridlock example

The figure above demonstrates the result if too many vehicles pass through certain sections of road at a given time. If this occurred during the simulation, then every vehicle except the ones stuck in the gridlock would finish. A method to stop gridlock from happening would be to reduce the quantity of vehicles being pushed onto the grid. We tried incremental pushing, where only a set amount of vehicles were pushed at a time. However, even though the degree of gridlock was reduced, we got the same phenomena with a large number of vehicles. It seems that it is not the size of the grid that manufactures the problem, but the type of road. Patterns were spotted when simulating the same map area that showed the same road intersections getting blocked:

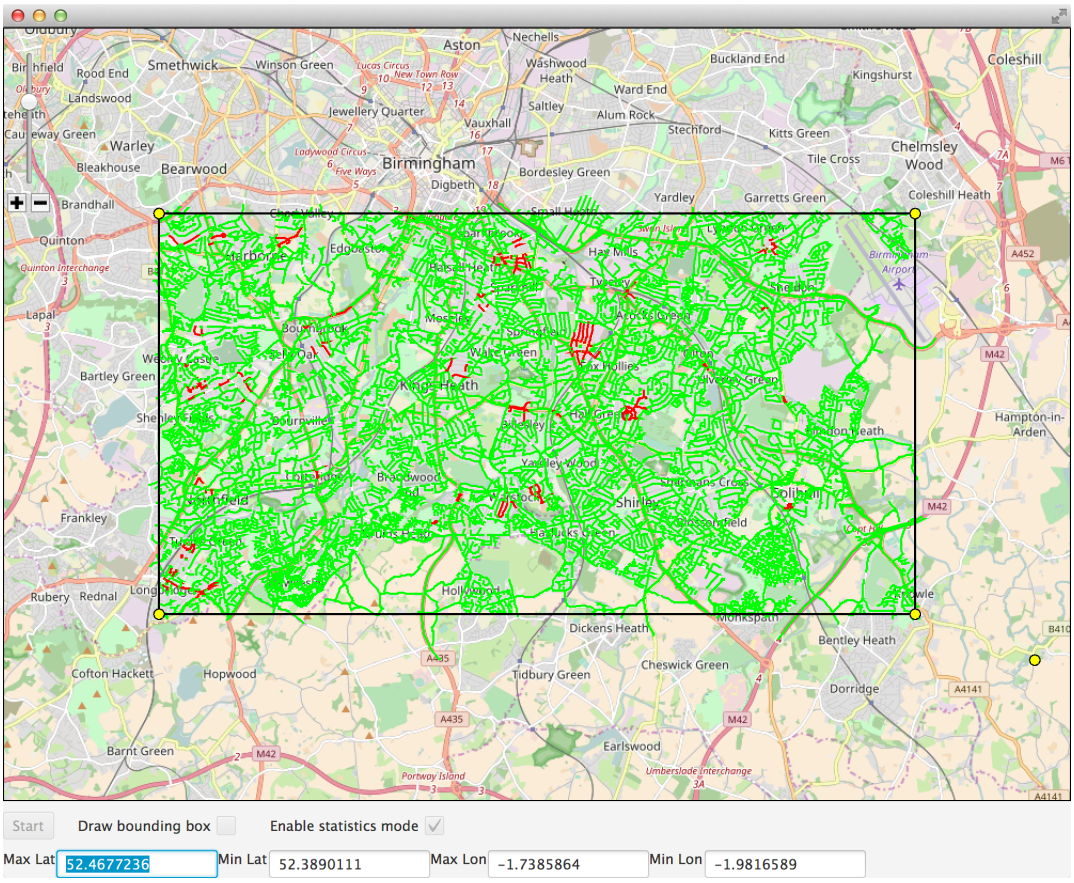

Figure 31: Gridlock with 5000 vehicles remaining

In the instance above, we put 75000 vehicles on a grid and the simulation ran well until about 5000 vehicles; the gridlock could have occurred earlier on in the simulation, but we only spot it when all vehicles that are not stuck have finished their journeys (the vehicle counter still decreases up until this point).

Total vehicles	Vehicle count at gridlock
50000	1243
60000	1658
70000	2883
80000	4718
90000	6475
100000	8744

Table 3: Scalability

The table above reveals the simulations scalability with large number of vehicles. The number of vehicles that get gridlocked is also present to make it apparent that the simulation is working. The table shows that the simulation has the potential to run a large amount of vehicles and for-fills our requirement of running a large-scale simulation.

6.1.5 Performance

As mentioned in 6.1.1, the performance of the simulation varied depending on the number of cars. During simulation, the map was responsive and the user can move the viewer position and zoom without delay. The most computationally intensive process comes when simulating large grids; creating the servers that join roads can take a long time for heavily populated map areas. During the execution of this method, the map hangs until all the connections are formed. The actual time to run a single simulation was very good at 10000 vehicles and below for medium sized grids:

Total vehicles	Simulation time (Milliseconds)
100	680
1000	1144
10000	1838

Table 4: Simulation time in milliseconds

Note that the above cars were simulated on the same grid and did not use the incremental push described in 6.1.4. The times do not include querying the database or server creation. Although the run time is dependant on the length of the vehicle's routes, the numbers above show that the simulation performs well given a large number of vehicles. All the tests were run on a MacBook Pro, which had a 2.13 Ghz Intel Core i7 and 4 cores; running on a faster machine could speed the times up even further.

7 Improvements

7.0.1 Traffic generation

Although our project was not focused on the generation of traffic, it was clear that to get realistic patterns in the data, it was dependant on this being somewhat accurate; pushing all vehicle onto the grid or incremental vehicle pushing could definitely be improved upon.

7.0.2 Validation

Despite the simulation exhibiting the patterns explained above, we were not able to validate the simulation output against real sensor data. Ideally, given realistic traffic initialisation, we could compare the number of vehicles received at an exit point in our simulation with a real-world sensor. This would give a better indication to the realism of the vehicle interaction and their journey times.

7.0.3 Visualisation

The visualisation was perhaps the element that was most lacking in the project. Our goal of representing individual cars was not for-filled and although we can see the changes in density, it is not as exciting. It did not help that we experienced a rendering bug either; a future consideration would be to change the map viewer source to something more robust.

7.0.4 Routing

As mentioned, we did not successfully implement proper routing in the system. The effect of this is that, although vehicles could traverse large area of the grid, their random routes were not as realistic and could create sparse areas of map, where no vehicles were assigned to go.

7.0.5 Signalised intersections

At the moment the intersections are not linked together by any sequence of ordering. It would beneficial for preventing gridlock if intersections had signals to notify whether the link could be entered (even if it was free). This way, no vehicles would get stuck waiting for another; this would also improve the scalability.

7.0.6 Parallelisation

In the simulation loop, each link and its vehicles are processed individually and per simulation step, a vehicle can only move one link. This means a vehicle has to wait until it gets accessed in the next loop iteration. It would speed up the simulation if the links were processed in parallel, with any shared data being handled using synchronization techniques.

7.0.7 Serialization

Although attempts were made to serialize the HashMap that stores the links and their connections, they were met with StackOverflow exceptions. Custom serialization is needed to be able

to persist the HashMap; if this could be stored for say a city, it would speed up the overall simulation time, as we would not have to wait for database queries and link / server creation (the most computationally intensive aspect of the system).

7.0.8 Map Source

Use OpenStreetMap proved useful for many things, such as node and edge collation. However, the consistency of its data was lacking in some areas. For example, lane and speed limit information was missing for large parts of the data; this meant some areas of simulation did not resonate with real life.

8 Conclusion

This project set out to create a traffic simulator that could simulate large quantities of vehicles for a given area of the world. We researched, selected and developed an existing model from the literature, building upon it to create an interactive system capable of running simulations on-demand. Our system allows the user to easily draw a designated section of map and let them visualise the simulation in action; this can either be performed in batch, or for single journeys. Through software engineering and implementation, we have for-filled most of the requirements that were laid out in the specification - the main omission was not being able to visualise the vehicles individually. Although our evaluation of the system showed it is not perfect in respect to certain simulation output, it definitely allows for future fixes and improvements; the most important of the ones listed in 7 being routing and traffic generation. Despite this, we are incredibly pleased with the simulation in general; in the limited time, we have matched and in some cases, exceeded the functionality of certain simulators that took many years to develop. We are also not aware of any simulator that has a shorter setup time, where by an area can be selected and the traffic simulated on it.

Appendix A Running the simulation

These are the steps that you need to complete in order to run the simulation.

1. Ensure you have the following dependencies installed
    - Java 8
    - Gradle
    - PostgreSQL
    - Git
2. Download the code repository from https://asb224@git-teaching.cs.bham.ac.uk/mod-60cr-proj-2016/asb224.git
3. Setting up the database
    - Create a database on your local PostgreSQL server (either through terminal or a GUI)
    - Run 'CREATE EXTENSION postgis;' for your database
    - Import the .sql file found in /Resources of the code structure using 'psql dbname < path/to/Resources/Birmingham.sql'
    - Start the server
4. Import the code as a gradle project (using the gradle files) into either IntelliJ (preferred) or Eclispe
    - In /src/Database/Query_V3.java, you must change the credentials (lines 31-33) to match your PostgreSQL details (including username, password and database name)
5. To run the simulation run the src/GUI/Display.java class containing the *main()* method
6. Once the GUI appears, you can:
    - Draw your own bounding box using the checkbox (followed by start button)
    - Enable statistics mode using the checkbox (followed by start button)
    - Run default location by pressing start button
7. Changing parameters
    - Change the number of vehicles on line 27 of src/Simulation/Simulate.java
    - Change the parameters in *increaseCars* (for statistic mode) on line 160 of src/GUI/Display.java
    - Change the incremental_push variable (number of vehicles pushed per step) on line 25 in src/Simulation/Simulate.java
8. Remember that choosing large amounts of vehicles for a small grid may result in grid lock

Appendix B Unit tests

Below are some examples of the units we ran:

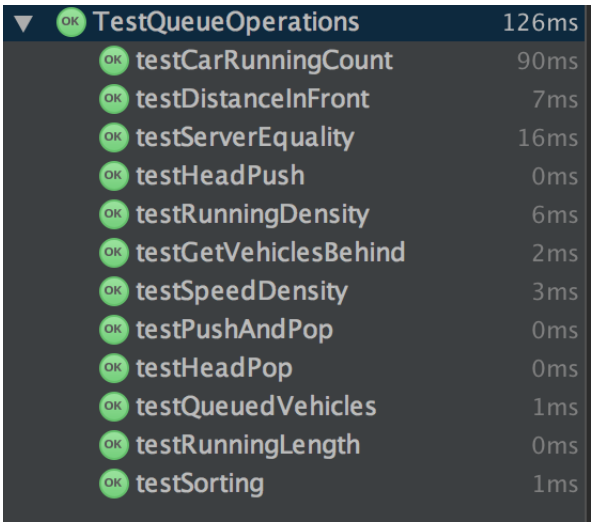

Figure 32: Test Queue Operations

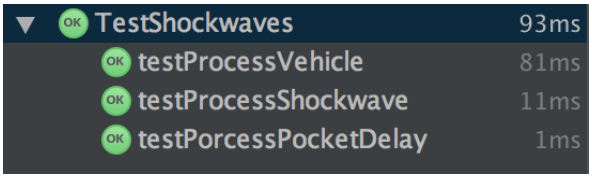

Figure 33: Test shockwaves

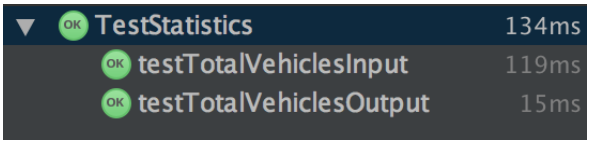

Figure 34: Test statistics

References

[1] Lighthill, M. J. and Whitham, G. B., *On Kinematic Waves. II. A Theory of Traffic Flow on Long Crowded Roads*, The Royal Society of London, 1178 (May 1955), 317-345.

[2] Richards, P. I., *Shock waves on the highway*, Operations Research, 1 (1956), 42-52.

[3] Nagel, K. and Schreckenberg, M., *A cellular automaton model for freeway traffic.*, Journal de physique I, 2(12), 1992, pp.2221-2229.

[4] Berto, F. and Tagliabue, J., *Cellular Automota, The Stanford Encyclopedia of Philosophy*, Metaphysics Research Lab, Stanford University, 2012, http://plato.stanford.edu/archives/sum2012/entries/cellular-automata/

[5] Rickert, M. and Nagel, K. and Schreckenberg, M., *Two lane traffic simulations using cellular automata*, Physica A: Statistical Mechanics and its Applications, 1996, 534-550.

[6] May, A. D., *Traffic flow fundamentals*, Prentice Hall, 1990.

[7] Whitham, G. B., *Linear and nonlinear waves*, John Wiley and Sons, 1974.

[8] , Payne, H. J., *Models of freeway traffic and control*, Mathematical Models of Public Systems, 1971, 51-60.

[9] Daganzo, C. F., *Requiem for second-order fluid approximations of traffic flow*, Transportation Research B, 29 (1995), 227-286.

[10] Aw, A., Rascle, M., *Resurrection of "second-order" models of traffic flow*, SIAM Journal of Applied Mathematics, 60 (2000), 916-938.

[11] Zhang, H. M., *A non-equilibrium traffic model deviod of gas-like behavior*, Transportation Research B, 36 (2002), 275290.

[12] Sewall, J. and Wilkie, D. and Merrell, P. and Lin, M. C., *Continuum Traffic Simulation*, Blackwell Publishing Ltd, 29 (2010), 439-448.

[13] Reuschel, A., *Vehicle movements in a platoon with uniform acceleration or deceleration of the lead vehicle*, Zeitschrift des Oesterreichischen Ingenieur-und Architekten-Vereines 95 (1950), 50-62.

[14] Pipes, L. A., *An Operational Analysis of Traffic Dynamics*, Journal of Applied Physics, 24 (1953), 274-281.

[15] Yang, Q. and Koutsopoulos, H. and Ben-Akiva, M., *Simulation Laboratory for Evaluating Dynamic Traffic Management Systems*, Transportation Research Record: Journal of the Transportation Research Board, 2000, 122-130.

[16] Burghout, W., *Hybrid microscopic-mesoscopic traffic simulation*, KTH, 2004, 185p.

[17] Wilkie, D. and Sewall, J. and Lin, M. C., *Transforming GIS Data into Functional Road Models for Large-Scale Traffic Simulation*, IEEE Transactions on Visualization and Computer Graphics, 2012, 890-901.

[18] Messmer, A., *METANET: a macroscopic simulation program for motorway networks*, Traffic Engineering and Control, 1990

[19] *SUMO Simulation of Urban Mobility*, http://sumo.sourceforge.net, 2016.

[20] *Statista*, https://www.statista.com/statistics/299972/average-age-of-cars-on-the-road-in-the-united-kingdom/, 2017.

[21] *The Society of Motor Manufacturers and Traders*, https://www.smmt.co.uk/2017/01/uk-new-car-market-achieves-record-2-69-million-registrations-in-2016-with-fifth-year-of-growth/, 2017.

[22] *PTV Group*, http://vision-traffic.ptvgroup.com/en-us/products/ptv-vissim/, 2017.

[23] *OSM4Routing*, https://github.com/Tristramg/osm4routing, 2017.

[24] *Java Hamcrest*, http://hamcrest.org/JavaHamcrest/, 2017.

[25] *OSM Download*, http://download.bbbike.org/osm/bbbike/Birmingham/, 2017.

[26] *GraphHopper*, https://www.graphhopper.com, 2017.

[27] *Java Streams*, http://www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html, 2017.