Dissertation Type: research

DEPARTMENT OF COMPUTER SCIENCE

MARMOSET

Multi-Agent Route Management using Online Simulation for Efficient Transportation

Alexander Hill

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Engineering in the Faculty of Engineering.

Friday 6^{th} May, 2016

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of MEng in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Alexander Hill, Friday 6th May, 2016

Contents

List of Figures

List of Listings

Executive Summary

Abstract

Since the mid-2000s, finding your way to a new destination has been as easy as entering your location and destination into an online service. Modern systems can take into account an enormous amount of data to provide the most efficient route. When provided with the start time of the journey, Google Maps is able to identify areas where there are levels of congestion, whilst Waze shares the location of their drivers to help improve the quality of this data.

However, these kinds of solutions make a core assumption about how routes should be returned. Each route is designed to be locally optimal for the user requesting it - but this means that routes may not be globally optimal for the system overall, creating congestion and traffic jams. This has been particularly problematic in large cities such as London, where 64,000 vehicles commute into the centre of town every working day.

At the same time, we are witnessing the rise of vehicles capable of autonomous driving. These vehicles will be more connected than any future vehicles, enabling new opportunites for improving the lives of drivers. With the additional information self-driving vehicles can provide to central systems, routing algorithms capable of finding globally optimal routes can be designed and implemented with ease.

However, we are still at least a decade away from self-driving vehicles becoming mainstream. Until then, we can research and understand the impact they will have by creating realistic simulations of vehicles driving on roads. Using these simulations, networked algorithms controlling massive fleets of vehicles can be designed, tested, analysed, and optimised without needing thousands of vehicles on the road.

This project has two core deliverables. The first is a simulation engine focused on city-scale vehicle simulation for the purpose of algorithm design. The second as a multi-vehicle routing algorithm designed to provide routes to an entire city of vehicles and create a reduction in the amount of congestion on the roads.

Summary of Achievements

For this dissertation I have:

- Researched the current state of the art multi-vehicle routing algorithms and simulation engines and techniques.
- Designed a powerful set of abstractions enabling fast experimentation and implementation flexibility for the routing algorithms and vehicle behaviour.
- Built and optimised a simulation making use of these abstractions.
- Designed a multi-vehicle routing algorithm based on insights from other algorithms as well as my own ideas.
- Implemented the algorithm on top of the simulation engine.

- Used the metrics from the engine to analyse and understand its strengths and weaknesses.

Supporting Technologies

This project has two parts - a back end engine written in Java, and a front end visualisation part running in the browser using JavaScript.

Back End

- OpenStreetMaps (OSM) [7] is used for the raw mapping data. Specific sections of the maps can be downloaded from Geofabrik.
- The Open Source routing engine GraphHopper [10] was used for performing basic routing requests and handling storage and processing of the OpenStreetMaps data.
- NanoHttpd [13] was used for a static file server to provide HTML, CSS and images to the front end.
- Java WebSocket [19] was used for communication between the front and back end.

Front End

- The map interface on the front end uses the JavaScript library Leaflet.js [1] for the map and marker APIs.
- Mapbox is used for the image tiles to display the underlying map.

Notation and Acronyms

OSM	:	OpenStreetMap
GraphHopper	:	Open Source Routing Engine, used for point to point routing requests
VANET	:	Vehicular-AdHoc NETwork
V2V	:	Vehicle to Vehicle
V2I	:	Vehicle to Infrastructure
SVG	:	Scalable Vector Graphics

Acknowledgements

I'd like to start by thanking the regulars of the back labs in 2.09 - James Burnside, Sam Healer, George Field, Hazel Doughty, Jamie Maddocks, Louis Auger, Andrei Ilisei, Ana Dumitras, Georgiana Ginghina, Freddie Caira and occasionally Ben Elgar for providing moral support and much needed distraction throughout the project.

I'm incredibly grateful to Benjamin Sach, who happily agreed to supervise a project with little more than a vague idea described in a single sentence. His boundless confidence in the project has always been a great source of energy and motivation.

The idea for this project came after a productive summer working at small startup called Pie Mapping. Max Glaisher's faith in my abilities and willingness to allow me to experiment with building a powerful routing platform gave me the knowledge and experience I needed to attempt a project focused on the excting fields of routing, transportation, and self-driving cars.

Much of my work at Pie Mapping was adding features to a fork of the GraphHopper routing engine, built and maintained by Peter Karich. Without his incredible technical skill and passion for open source (let alone his willingness to provide free support to GraphHopper's users) this project would not have been possible.

I would also like to thank my parents for their unwavering support and faith in my abilities.

Chapter 1

Contextual Background

1.1 Routing in the 21st Century

When MapQuest first launched in 1996, most drivers found their way to new locations using physical paper maps. MapQuest was the first mainstream online service to change that, allowing users to enter their location and destination and have a route provided to them. Using these routes required printing out or writing down the instructions, meaning changes to the driving environment (such as traffic or roadworks) could not be anticipated.

In 2005, the first online version of Google Maps was released. Initially, Google Maps lagged behind MapQuest, but with rapid improvements to their map and route quality they soon took the lead. Google was quick to improve their mapping and routing offering, integrating real-time traffic analysis to give its users a significantly better experience. However, for a time most routes were still printed - even vehicles with GPS navigation rarely used non-static mapping information.

It took the release of the iPhone in 2007 to take full advantage of the new information available - suddenly, people were able to plan routes and modify them during the journey, whilst Google could use this information to improve their data on congestion.

Google weren't the only company with this idea - a small startup called Waze had a similar vision. Their app provided routes to users that avoided traffic by tracking the movement of other vehicles using the app. If cars in a certain area were stationary, the app informed other drivers that there may be congestion and offered them alternative routes. Their strategy proved successful, and in May 2013 they were acquired by Google for an estimated $1.3 billion.

1.2 Congestion

The proliferation of information whilst driving is particularly important in cities, where congestion plays a primary role in determining optimal routes. Cities also struggle with 'rush hours' at the start and end of the working day, when congestion is at its worst. In London, this issue was so problematic that in 2003, a congestion charge was put in place to deter drivers from entering and leaving the centre of the city at peak times. In spite of this, upwards of 64,000 people drive into the centre of London every day [6].

Looking forwards, we see two global trends that suggest that congestion will become more of an issue in future. Firstly, increasing numbers of people are living in cities - even with public transportation, this increases the number of vehicles on the road notwithstanding the decrease in the proportion of households owning cars. Secondly, on-demand transport solutions such as those provided by Uber are placing more vehicles on the road, especially at peak times. One of Uber's key insights is using market techniques to match supply and demand better than existing taxi and minicab services. If there is an increase in riders requesting vehicles in a certain area, Uber activates "surge pricing", increasing the cost of the ride by a fixed multiple - for example, 1.5x. This information is sent to their network of drivers, who will move to the location in search of higher fares. The net result is that high demand in certain areas creates further congestion, despite the efficiencies on-demand transportation has over individual car ownership.

1.3 Self-Driving Vehicles

Vehicles have improved alongside the development of effective routing algorithms; GPS based navigation, with traffic information and turn by turn routing has now been integrated into most new vehicles available to consumers. At the same time, we are beginning to witness the rise of self-driving cars, capable of planning and executing routes themselves with no need for human intervention. This is being approached from two different perspectives.

The first is through the modification of existing cars - adding features such as cruise control, automatic braking systems, reading of road signs, lane following on highways, and other features for automated control until the car will need minimal or no human intervention. This approach is being taken by Tesla, who have released multiple software updates to their car software that improves the car's ability to drive itself. The software is not yet capable of full autonomy, but each update brings self-driving cars closer to reality.

The second approach is from companies like Google, who have been building a self-driving vehicle 'from scratch'. Their vehicle has no steering wheel and requires no human intervention to drive from one location to another. Google's car has driven more than 1.5 million miles with fewer accidents than a human driver would have had.

This raises many questions about the role of personal transportation and the effect this will have on congestion. Will driving become as dated as horse-riding is today, or will people's desire to drive mean that there will always be human driven vehicles on the road? More importantly, will self driving cars improve or worsen congestion? Furthermore, how will these vehicles decide what routes they should take?

Answers to these questions are important to many people, including individual drivers, companies owning large fleets of vehicles, and city planners. Although it is not possible to provide definitive answers, simulation provides a technique for evaluating how future transportation will look and the impact it may have.

1.4 Simulation

The simulation process of modelling, making assumptions about behaviour, and analysing the results has proven to be effective in numerous situations. Vehicle simulation can be used in a number of ways:

- Simulating current vehicle behaviour and traffic conditions to identify improvements to the road network.
- Modifying the road and transport networks (such as busses or taxis) in the simulation to identify the impact changes would have.
- Designing and running novel algorithms for simulating self-driving vehicles, modelling their behaviour in response to both human drivers and other self-driving vehicles.

1.4.1 MATSim

One such simulation tool is MATSim, short for Multi-Agent Transport Simulation. MATSim is commonly used by researchers in the intelligent transport field for macroscopic (large-scale) simulations. MATSim can be used to perform many different types of simulation, from routing air traffic to planning for evacuation in crisis situations. This flexibility is of course an advantage - but it comes at a cost. As the engine is so open-ended, it requires substantial setup and is complex to use for vehicle routing - there is no built in support for OpenStreetMap data, instead requiring a multi-stage process to convert the data into the right format. Worse, a separate proprietary tool is needed to visualise the routing information.

This complexity and inaccessability of data makes it harder to iterate on new ideas. If the data were easier to visualise, an algorithm's flaws could be identified near instantly and suitable modifications made immediately. Unfortunately, two of the more popular MATSim visualisation tools suffer from the same problem as MATSim - they are designed to support a massive variety of use cases, meaning they aren't especially good at any specific one of them.

Via is a commercial application for visualising the output of MATSim simulations. It is capable of providing a large amount of information about a completed simulation, but is also able to import tracked vehicles from GPS data and perform analysis on that. Of course this has its use, but it also has the downside of obscuring the simple use case of watching and understanding the behaviour of a vehicle simulation.

A brief attempt at a basic simulation highlighted the complexity and struggle of the MATSim workflow - specifically, the confusing and unintuitive user interfaces of the tools required, the necessity of installing multiple pieces of software (none of which directly reference each other) and the surprising difficulty of viewing data on a map.

1.4.2 SUMO

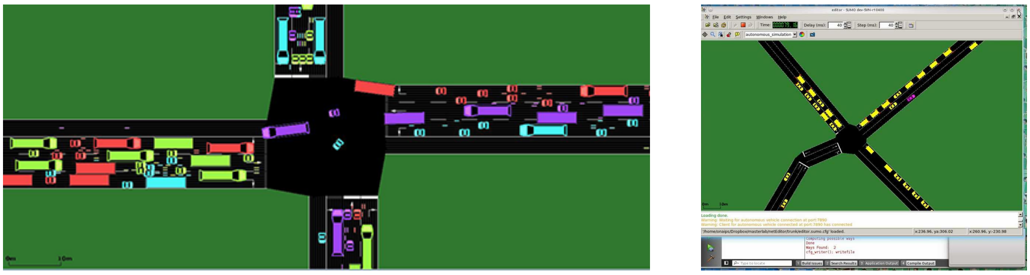


Figure 1.1: Screenshots of the SUMO simulation engine

On the other end of the spectrum, we have SUMO, short for Simulation of Urban MObility [12]. SUMO is designed for microscopic simulations, down to individual vehicles and traffic lights. Unlike MATSim, SUMO is a more focused and unified project (although the default distribution comes with nine pieces of software), primarily enabling city planners to see the impact of changes to the road network. Bus timetables and schedules could be changed, traffic lights added or modified with various models of vehicle behaviour.

SUMO seems to lack many of the issues with MATSim - it is mostly unified, designed for a specific use case and makes it easy to visualise vehicle behaviour, as shown in Figure 1.1. However, SUMO's use cases require microscopic simulation, meaning it is very accurate for simulating the behaviour of each vehicle, traffic light, pedestrian and truck. Unsurprisingly, this comes at a substantial performance cost. SUMO's accuracy and focus makes it a good tool for its use case, but it does not lend itself to the city scale simulation and in-depth data analysis that is needed to answer questions about the future of transportation.

1.5 Objectives

Unlike existing simulation engines, this project aims to create a system designed exclusively for city-scale real-time macroscopic car simulation, for aiding development of new multi-vehicle routing algorithms. By focusing on this use case, we believe a tool can be created that surpasses existing simulation engines in both performance and ease of use.

> This project aims to have two key deliverables. The first is a flexible, fast and easy to use simulation and visualisation engine. The second is a multi-vehicle routing algorithm designed and developed using the simulation engine. The concrete objectives are:
>
> 1. Research existing algorithms and simulation tools to identify the strengths and weaknesses of current approaches.
> 2. Design a simulation architecture that allows for fast experimentation, easy integration with real world information and full implementation flexibility.
> 3. Build and optimise the simulation engine on top of existing open source tools.
> 4. Design and implement a novel multi-vehicle routing algorithm.
> 5. Use the tools provided by the simulation engine to analyse, understand, and optimise the routing algorithm.

Chapter 2

Technical Background

This chapter contains both the algorithmic and practical details that are required to understand how the project will be delivered. The algorithmic section covers existing single and multi-vehicle routing algorithms as well as the processes used for realistic vehicle simulation. The implementation section introduces the open source software that provides routing algorithms and road network information to the simulation engine.

2.1 Algorithmic Background

2.1.1 Dijkstra's Algorithm

Dijkstra's Algorithm [5], originally designed in 1956, forms the foundation of most modern routing algorithms.

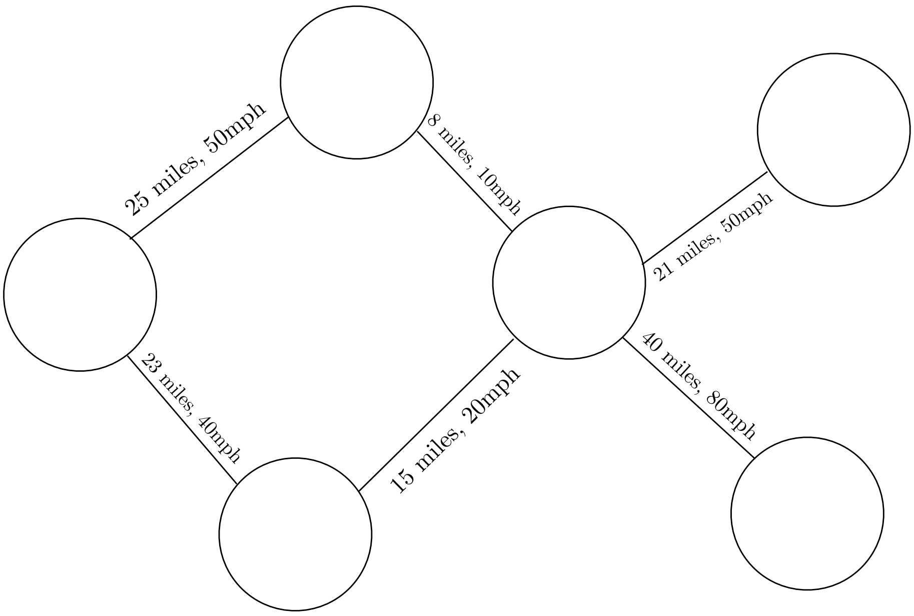


Figure 2.1: Graph showing roads with distance and speed limit at the edges

The algorithm operates on a graph consisting of nodes connected to each other by edges. Each edge has a weight, representing the cost of travelling along that edge. Given a source node S and a destination node D, the algorithm returns a list of edges representing the shortest path between S and D.

At its core, the algorithm picks the edge with the next shortest distance from the source node and updates the minimum distance value for that node. It repeats this process until it finds the source node, then works backwards to construct the shortest path between the two nodes.

There are a few key ways of implementing the algorithm, making use of different types of heap. They each have different performance characteristics, but the most common approach uses a binary heap and offers $O(E \log V)$ performance, where E is the number of edges and V is the number of vertices in the graph.

When routing in the real world, the weights of the graph are not usually fixed. As shown in Figure 2.1, the road's speed limit and distance are stored on the edges of the graph. This data is used in combination with information about the vehicle to calculate the travel time, which can then be used as the weight of an edge. For example, unlike cars motorcycles may not be able to reach 80mph, so we would use the maximum speed of the vehicle rather than the road to calculate the travel time for that edge. This concept allows a single graph to provide routes to multiple types of traveller.

Bidirectional Dijkstra's Algorithm

By default, the algorithm searches outwards from only the source. We can improve the performance of Dijkstra's algorithm by searching backwards from the destination and forward from the source simultaneously. This has a dramatic effect on the number of edges that are searched, as can be seen in Figure 2.2a.

A* Algorithm

By itself, Dijkstra's algorithm will search all edges based on their cost from the start node. However, when routing on a real world map this leads to a lot of wasted work searching away from the destination - this can be seen in Figure 2.2a.

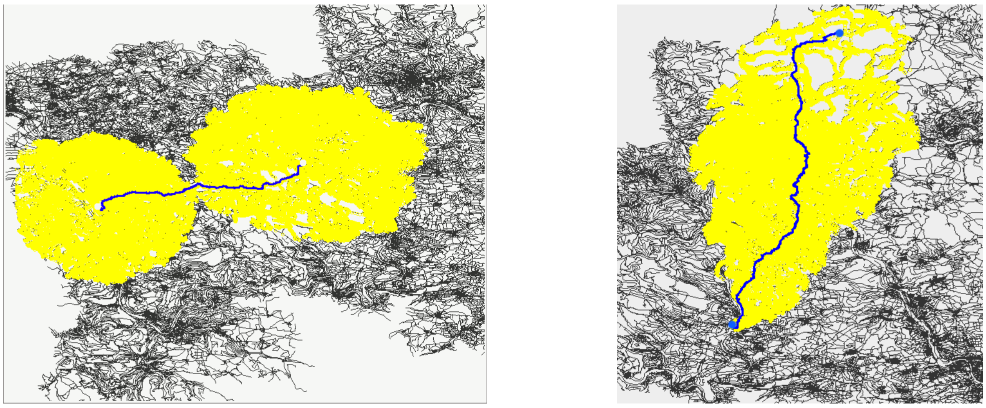
Dijkstra's algorithm picks the next edge to check based only on the distance from the source node, using this as the key in a priority queue. The A* algorithm modifies this by including a heuristic function representing an estimate of the distance to the destination node. This allows it to prioritise nodes that are more likely to route towards the destination. Figure 2.2b shows the paths searched by the A* algorithm - note that it does not look as far behind the source node as Dijkstra's algorithm, to its left.

A bidirectional version of the A* algorithm can also be used for further improvements to performance. However, although a bidirectional Dijkstra's algorithm is significantly more efficient than a single direction one, the same is not true for the A* algorithm as the heuristics cannot be as tight [9].

Contraction Hierarchies

If the weights on each edge are known in advance, we can pre-process the graph to compute the shortest path between certain nodes and improve the performance of routing requests by orders of magnitude. Contraction Hierarchies [8] is one technique for graph preprocessing. It works by building shortcuts on top of each other in a hierarchy, and using a modified bidirectional Dijkstra's algorithm that only uses shortcuts higher than the current one to route between two points.

Although they double memory usage, in practice Contraction Hierarchies have a huge impact on query times. For routing from Moscow to Madrid, any Dijkstra based algorithm takes at least 10 seconds, compared to less than 0.05s for a processed graph [11].



(a) Bidirectional Dijkstra's Algorithm (b) A* Algorithm

Figure 2.2: Paths searched by each algorithm

2.1.2 Multi-Vehicle Routing Algorithms

Providing viable routes for a single vehicle is a problem with accepted solutions and various implementations used in many successful commercial products. However, the problem of routing multiple vehicles simultaneously has many solutions [23, 16, 2, 26, 4, 15, 22], but no accepted 'best practice' technique.

This is primarily due to their research focussed nature - it is not currently possible to route all vehicles collaboratively, so many types of algorithm exist to deal with different expectations of the future of transport communication.

As such, multi-vehicle routing algorithms usually fall into one of two categories. Some algorithms are designed to have a large scale overview of where each vehicle is on the road using some secondary infrastructure, and will dispatch routing information to the vehicles centrally. On the other end, algorithms may assume no central information and instead rely upon vehicle to vehicle connections for passing information. Algorithms relying on the installation of additional infrastructure are referred to as **V2I**, standing for Vehicle to Infrastructure, whilst **V2V** refers to Vehicle to Vehicle algorithms. A simulation engine capable of implementing both V2V and V2I communication models is known as a V2X engine.

VANET

A VANET (Vehicular-AdHoc NETwork) is a high-level use of V2V communication. Although primarily theoretical at the moment, wireless networking protocols and algorithms have been designed to support the creation and use of a VANET. VANETs have a number of potential use cases - allowing vehicles to follow one another with no driver intervention, real-time calculation and distribution of traffic information and so on. V2V algorithms relying on a VANET have been created and shown to offer improvements in routing performance.

BeeJamA

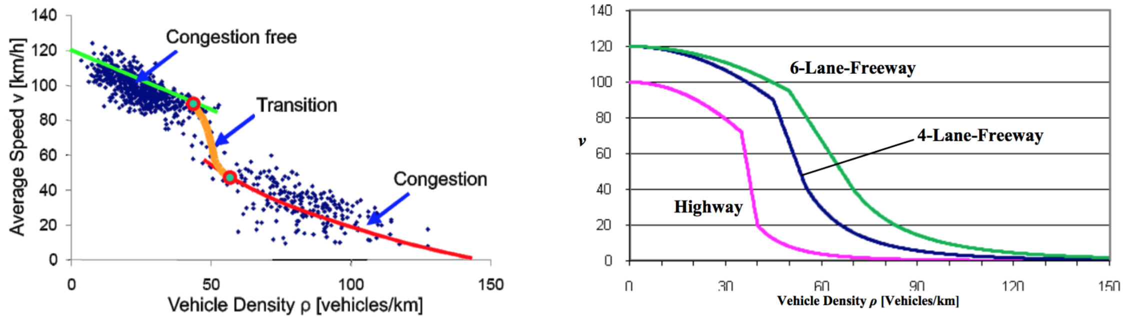
The BeeJamA [23] algorithm is a V2I approach for multi-vehicle routing. It is based off an algorithm originally created for routing in packet switching networks that has been adapted for use on road networks. Its behaviour is inspired by the behaviour of bees, which are able to effectively search and scavenge for food in a large area surrounding their hive.

The map is split into regions called Foraging Regions. Each region stores two types of table - an Intra-Foraging Region routing table and an Inter-Foraging Region routing table. These store the cost of using a particular route to reach the destination. The tables are kept up to date by sending virtual vehicles between regions, which record how long their journey took and update the tables of the region they have entered.

Whilst routing, each vehicle consults the routing table and probablistically picks its next step. This is a key part of the algorithm that helps keep routes uncongested - if every vehicle took the optimal path, it would no longer be optimal as it would have high levels of congestion. Results from a custom-built simulation engine have shown that the algorithm can perform better than using Dijkstra's algorithm with delayed traffic information.

When routing with Dijkstra's algorithm, the travel time for each edge is usually estimated based only on the road's speed limit and distance (as well as the maximum speed of the vehicle). The BeeJamA improves upon this drastically by taking into account the number of vehicles on a given road. Instead of using the maximum velocity for a road, BeeJamA uses a congestion function that takes into account the density of vehicles on a given road as well as the road type. The congestion functions they and other researchers have derived can be seen in Figures 2.3a and 2.3b.

Empirical analysis of real world data has been performed with the goal of creating a function from density of vehicles to velocity [25, 20].



(a) Empirically derived congestion function from [25]

(b) Mathematical congestion functions for different types of road from [23]

Figure 2.3

2.1.3 Nagel-Schreckenberg Model

The Nagel-Schreckenberg Model [14] is a cellular automaton model for the flow of traffic on roads.

The model splits roads into discrete cells, with each car taking up a single cell at a time. The vehicles then follow 4 rules to simulate the flow of traffic. Each car has a fixed velocity v, which represents the number of cells the vehicle intends to move forward. The first three steps determine the velocity, with the final step updating the position of each vehicle. The model does not allow for overtaking, exhibiting realistic behaviour for traffic jams and flow.

1. **Acceleration** - if the vehicle is not at the max speed and there is enough space ahead, increase the velocity by 1.
2. **Slowing down** - if there is a vehicle nearer than the current velocity, reduce velocity to one cell less than the distance to the vehicle in front.
3. **Randomisation** - reduce the velocity by 1 with probability p.
4. **Movement** - move each vehicle forward by its velocity v.

Each cell is meant to approximately represent the size of a single vehicle, whilst each step (running through all four rules) represents a discrete amount of time - usually a single second.

We'll now run through a brief example of the effects of each step. Our road will have nine cells and two vehicles. Our max speed (v_{max}) will be 5. For this example, we will ignore the randomisation step.

	4				0			

We first perform the acceleration step for each vehicle. The blue vehicle is currently at speed 4 - as the red vehicle is only 3 cells away, it does not increse its velocity. The red vehicle increments its speed from 0 to 1.

During the slow step, the blue vehicle must reduce its speed from 4 to 3 so it does not hit the red vehicle.

				3		1		

The vehicles now move to their new destinations. This demonstration briefly shows the interaction between two vehicles, but does not demonstrate the creation and flow of traffic jams. In Figure 2.4, we can see how traffic jams form and move in the Nagel-Schreckenberg model. This graph shows what happens when there is a density of 0.1 cars per cell.

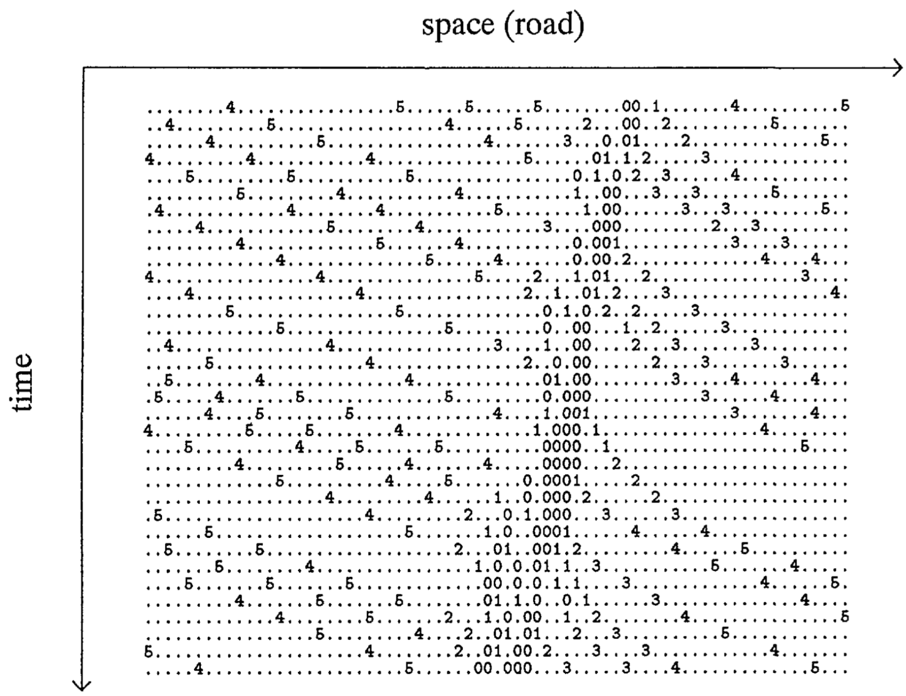


Figure 2.4: Formation of traffic jams in the Nagel-Schreckenberg Model from [14]

2.2 Implementation Background

2.2.1 OpenStreetMaps

In OpenStreetMaps, there are three main elements - nodes, ways, and relations. Each of these can have certain tags that describe the real-world object they represent. For example, a way could have tags describing it as a highway with 3 lanes, whilst a node could have tags identifying it as a park bench or phone booth.

A standalone node is usually a single entity with a latitude and longitude as well as an ID. By themselves, they can be used to represent small features such as traffic lights, lampposts, or pylons. However, a way is also defined by an ordered list of nodes.

Ways are the most important element in OpenStreetMaps. If a way starts and ends at two different nodes, it is called an 'open' way. This is commonly used for sections of roads and paths. A way that starts and ends at the same node is called a 'closed' way. Often this is used to represent an area - such as a park or the shape of a building - but can also be used for roundabouts and circular barriers. There are tags for identifying if a closed way is an area or not (primarily the `area=yes` tag, but many things are defined as areas even without this tag).

Finally we have relations. These are the most complex type of element, holding an ordered list of ways, nodes and other relations and have tags for describing the relationship between them - for example, a bus route could be described as a list of ways and nodes. For the purposes of routing, it is only important to know that relations are used for turn restrictions - the rules that define which directions a vehicle may move from one road onto another. This is important for routing much more than mapping, as users would be frustrated to find that a route they had expected to travel on is illegal or unsafe in practice.

2.2.2 GraphHopper Routing Engine

GraphHopper has been built to be fast, flexible and powerful. It covers the full flow of creating a custom routing service - from parsing and importing OpenStreetMaps data, processing the network for performance improvements, routing using multiple Dijkstra and A* algorithms and running a web server for making requests. Additionally, it has built in support for car, bicycle, pedestrian and other types of travel as well as the ability to create custom vehicle types.

Figure 2.5 shows the salient parts of the architecture for this project. Note that we are not using the web or android modules, so no details regarding their functionality has been included. For the most part, the use of routing will be relatively high level - simply requesting a list of edges between two points. However, the underlying engine also has a number of features that make it easy to customise, which will be used for more complex situations.

It is important to note that the conversion from OSM data to GraphHopper data is done only once (as it is a time consuming process) and stores its results on disk.

OpenStreetMap Encoding

The first thing to understand is how GraphHopper converts the OpenStreetMap data into a graph that can be used for routing. For the purposes of routing, we are primarily concerned with ways that represent roads. However, a way does not represent the entirety of a given road - but is also not granular enough to allow routing. Figure 2.6 shows the difference between ways, roads and edges. We can see that the physical Holloway Road extends beyond the OSM way (red). However, if we used the way for routing we could not route from Hargrave Road to St John's Villas via Holloway Road. To perform routing, GraphHopper splits every way into an edge whenever there is a junction. GraphHopper's edges can be seen in orange. These can be used to perform the routing calculation mentioned above - there is now an edge along a subsection of Holloway Road that connects Hargrave Road to St John's Villas.

Flag Encoders

Internally, GraphHopper compresses the information about each edge into a single integer - so each edge takes up no more than 32 or 64 bits.

The `FlagEncoder` interface (and the `AbstractFlagEncoder` class) define what operations have to be done to convert a section of a way into an edge. Edges store a few key pieces of information - by default, GraphHopper stores the length, speed and ID of an edge, with additional information optionally stored by the flag encoder itself.

For example, the `CarFlagEncoder` uses the type of road to calculate the correct maximum speed of the vehicle and then sets the speed for the edge, using 5 bits. The `MotorcycleFlagEncoder` can do the

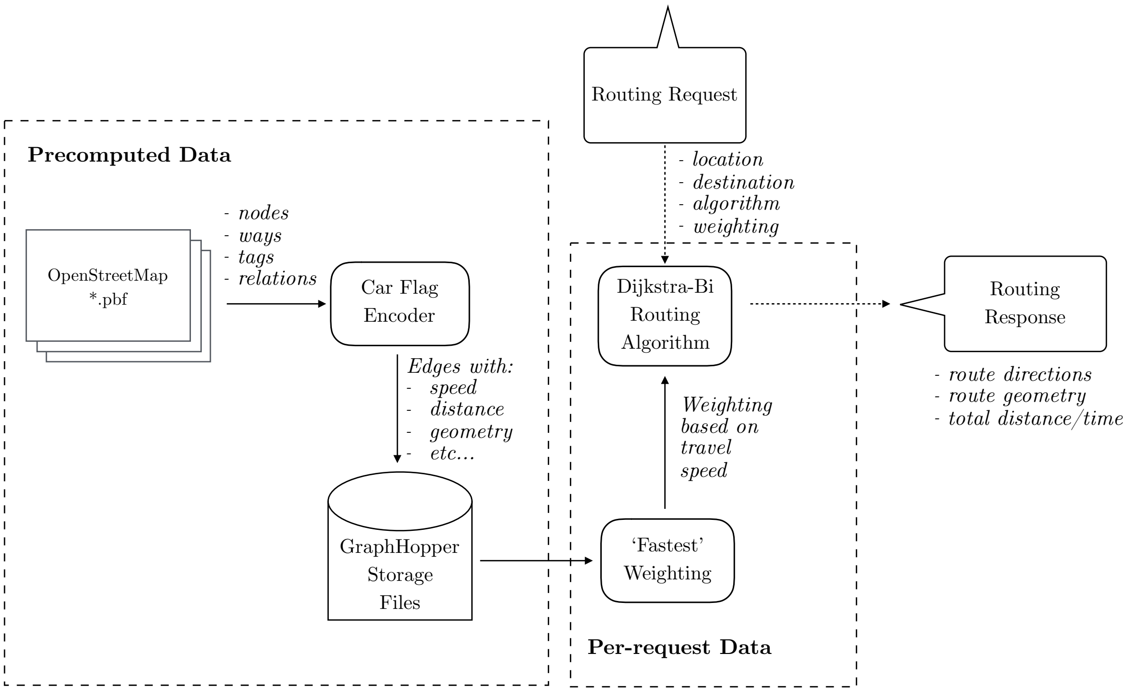


Figure 2.5: An overview of the GraphHopper Architecture

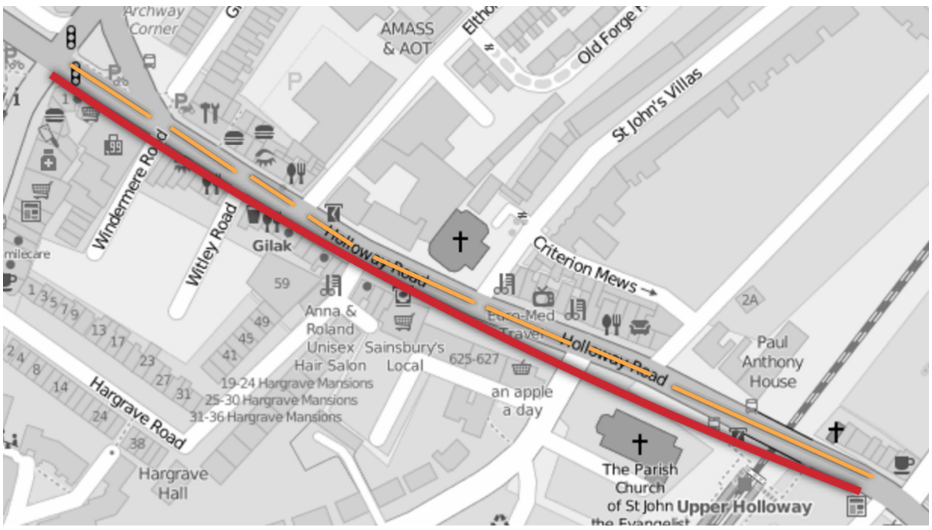


Figure 2.6: OpenStreetMap way representation compared to edges in GraphHopper

same thing, but with different speed values for each type of road. The concept can be extended to storing even more information at each edge - for example, 3D graph data can be stored for bike and walking routes, whilst a FlagEncoder for trucks could store height, weight, width and length restrictions to ensure safe travel.

Weightings

Flag Encoders are used to define what information can be used whilst routing - but this data is fixed once the OSM file has been processed. A weighting defines how the data stored at the edges is used. For example, one can search for the shortest route, the fastest route, or a custom weighting that incorporates other information stored in the edge.

Routing Requests

As it is primarily designed to work with its web module, GraphHopper uses a request-response pattern for requesting and receiving routes.

```
GHRequest ghRequest = new GHRequest(startLat, startLon, endLat, endLon);
ghRequest.setWeighting("fastest");
ghRequest.setAlgorithm("dijkstrabi");
GHResponse ghResponse = graphHopper.route(ghRequest);
```

Listing 2.1: GraphHopper request and response

The `GHRequest` class stores all the information required to make a routing request, including the weighting and algorithm to use. An example of its use can be seen in Listing 2.1.

The response is returned in a `GHResponse` class, which holds information about the route in a number of different ways. The user is also responsible for checking the `hasErrors` method before requesting the route itself. Once this has been done, the route itself can be read.

As GraphHopper supports the use of an alternative routes algorithm (meaning a single request can respond with more than one route), the GHResponse class has both a `getAll` and `getBest` method, which return a list of `PathWrapper`s and a single PathWrapper respectively. The PathWrapper holds the route itself as well as some metadata. The route can be accessed as a list of instructions or as the raw list of points for visual display. Additionally, the total distance and time of the route are recorded.

However, GraphHopper does not provide a list of edges or of OSM Ways that are included in the route. Internally, the list of edges is calculated with the `calcPaths` function, which returns a list of `EdgeIteratorState` objects. These objects are fundamental to the way the graph is stored an accessed in GraphHopper.

GraphHopper uses the flywheel pattern for efficient access to graph data. The EdgeIterator and EdgeIteratorState class are the primary way of interacting directly with the edge data. EdgeIteratorState is an interface containing getters and setters for the edge ID, its geometry, the flags stored by the FlagEncoder, and the name of the road the edge is a part of. It also allows access to the nodes at either end of this edge. The node at the start of the edge is called the Base Node, whilst the node at the end of the edge is the Adjacent Node.

Chapter 3

Project Execution

3.1 Overview

The execution of this project was split into 4 stages, each with its own goal.

1. **Initial Implementation** - create the initial client and server with vehicles moving on a map.
2. **Realistic Simulation** - implement the Nagel-Schreckenberg model to make the vehicle behaviour more realistic.
3. **Performance and Architecture Improvement** - make the simulation engine fast and adaptable enough to deal with the challenges of algorithm design.
4. **Algorithm Design and Implementation** - create and improve a novel approach to the multi-vehicle routing problem using the simulation engine.

3.2 Initial Implementation

The initial implementation had a few key goals, with a primary objective of showing basic vehicles moving on a map. This stage was essentially a way of setting up the basic architecture of the project, without finalising the details of how routing algorithms would be implemented.

3.2.1 Architecture Design

Figure 3.1 shows a simple architecture diagram for the initial implementation of the engine. Once the client has made a request, the engine, sets up and starts routing the vehicles. Each vehicle is responsible for storing and updating its own position on each timestep. This data is converted into a string and sent to the client, which then moves each vehicle on the map.

One of the goals of this project was to make it possible to visualise and simulate the vehicles at the same time. This naturally lends itself to a client-server architecture, and with the rate of improvement of modern web browsers it seemed like a wise choice to use a browser for the visualisation component.

This also means that the server can be run remotely with no additional setup - more computationally expensive algorithms could be run on high-powered servers with the results still visible locally for the user.

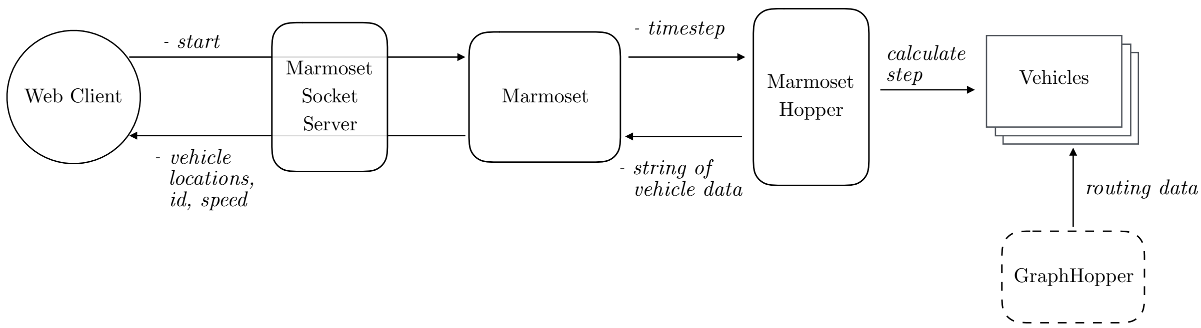


Figure 3.1: The initial architecture for the simulation engine

3.2.2 Project Setup and Modularisation

When setting up the project, it was important to be able to use and modify the underlying GraphHopper routing engine. The GraphHopper project is setup with four existing modules - `core`, `tools`, `web`, and `android`. Although it would have been ideal to use GraphHopper as an external library, it was necessary to make minor internal changes to the main engine. As such, a GitHub fork of the project was created with `marmoset` as a fifth module in the project.

GraphHopper uses the Maven dependency and build tool, with its own custom shell script for building, running and testing different versions of the engine. To keep Marmoset separated from the core project, a custom shell script for building and running the Marmoset engine was created. It supports four main actions - clean, build, rebuild, and run. It also allows multiple commands to be run in succession for convenience.

Additionally, it was desirable to create a modern codebase in spite of the core engine being written in Java. Although Scala was briefly considered, much of my work would rely on extending and using the existing Java APIs in GraphHopper. Thankfully, Java 8 has introduced a number of key tools that allow functional-style code to be written. The code below shows the difference for performing a simple task - converting a list of objects into strings and joining them by commas.

```java
public String getVehicleData()
{
    StringBuilder sb = new StringBuilder();
    for (VehicleController v : vehicles)
    {
        sb.append(v.getVehicle().toString());
        sb.append(",");
    }
    // remove last comma
    sb.deleteCharAt(sb.length() - 1);
    return sb.toString();
}
```

Java 7 Implementation

```java
public String getVehicleString()
{
    return vehicles.stream()
        .map(Vehicle::toString)
        .collect(Collectors.joining(","));
}
```

Java 8 Implementation

Here we can see how six lines of code can be condensed into a single, more readable line using the Java 8 Stream API.

3.2.3 Server Implementation

For the raw file server, the NanoHttd [13] library was used, as it requires minimal setup and is easy to run on a separate thread. For the client-server communication, initially the NanoHttpd WebSocket implementation was used, but it did not appear to be fully functional. As such, the Java-Websocket library [19] was used instead, extending the built in WebSocketServer class to create the `MarmosetSocketServer` class.

In the architecture diagram (Figure 3.1), we see four main classes on the back-end.

The `MarmosetSocketServer` class handles the connection between server and client. It keeps track of each of the connected clients and offers a simple command to distribute vehicle position data to each of them.

The `Marmoset` class is a static class, and is the main entry point for the program. It initialises the file server and WebSocket server, and creates a new thread for running the `MarmosetHopper` timesteps. It also handles passing data from `MarmosetHopper` to the socket server.

In most types of simulation, time must be split into discrete steps that represent a fixed time interval in the real world. Looking at Figure 3.1, we can see that it is the Marmoset class that triggers each timestep. At this stage of the project, it had not been established how effectively the front or back end would perform at receiving and rendering the vehicles. As a way of simplifying the data flow and processing, the Marmoset class simply waits one second between each call of the timestep function. As both front end and back end take significantly less than a second to perform their tasks, this was an appropriate simplification for this stage of development.

The `MarmosetHopper` class holds the list of vehicles and creates an instance of the GraphHopper routing engine. It initialises all the vehicles, instructs them to update on each timestep, and gathers their position information together to be sent to the clients. When initialising the vehicles, the starting location and destination are chosen as random latitude and longitude co-ordinates within London.

Finally, the `Vehicle` class represents a single physical vehicle on the map. Each vehicle holds its location and the route it plans to take. The routing information is obtained by requesting a route from GraphHopper. The route is returned in a number of ways, including as a list of points that can be used to draw the route. This does not include the speed of each road travelled, so cannot be used for realistic simulation. However, for this initial implementation the location is updated on each timestep by simply moving to the next point in the list. This is not realistic, but it does allow us to verify the functionality of all parts of the system without touching the details of routing. The vehicles return their location as a string containing their ID, latitude and longitude.

3.2.4 Client Implementation

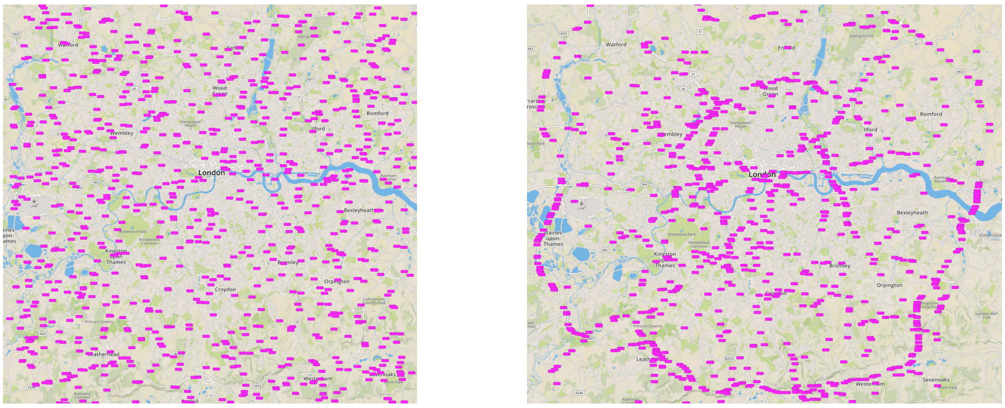
When loading the web page, a map is created and centred on central London. The client then connects to the WebSocket back end and listens for data. When it receives the vehicle data, it creates or updates the position of each vehicle marker.

The Leaflet.js [1] library is used for both map and marker creation. A simple `Car` class has been created to keep track of the location of each marker. It stores a reference to the Leaflet.js Marker object and provides a method to move the marker to a new location. The Leaflet.AnimatedMarker [17] library handles smoothly moving the points to their next location, making the cars look like they are driving around the map in real-time.

Meanwhile, a single `CarSet` object connects to the WebSocket server and stores the `Car` objects. When data is received, it creates new `Car` objects or updates the position of existing vehicles using their `moveTo` method.

3.2.5 Results and Improvements

At the end of this section of the project, the engine was capable of performing its core task of simulating and visualising vehicles. In spite of many of the simplifications in the system, it provides quite realistic results - 3.2b shows that even after a small number of iterations certain roads become more congested than others. This matches the reality of driving on London roads - the M25 and North Circular suffer frequent delays due to high levels of congestion.



(a) 10th Iteration

(b) 200th Iteration

Figure 3.2: Vehicle positions at different iterations

However, there are many core features missing from the engine at this point:

- The engine simulates a fixed number of vehicles (1000); no more can be added or removed.
- Vehicles do not move realistically; as they simply jump to the next point on their route, a short curved path would take longer to travel along than a long but straight motorway.
- Vehicles are independent, travelling on their own with no sense of traffic or congestion.
- The timestep is fixed to one second, even though the actual processing takes substantially less time for both front and back end.
- Metrics are not tracked; other than watching the simulation, there is no further understanding that can be gained from simulating.
- The simulation cannot be paused without terminating it and starting again.

The focus for the next stage of develeopment was on improving the realism of the vehicle simulation.

3.3 Realistic Simulation

In section 2.1.3 we introduced the Nagel-Schreckenberg Model for traffic flow simulation. This section discusses the implementation of this model on top of the existing simulation engine described above.

The original model is designed for a single road, either in a loop or an extended straight stretch. Implementing this on a road network introduces some additional challenges, particularly when handling the integration between GraphHopper and OpenStreetMaps.

The two main concerns are how the cells should be stored and how the cells should be used in conjunction with the routes created by GraphHopper.

3.3.1 Cell Storage

A number of techniques for storing the cells were considered. Firstly, we must consider how we create edges. We have the option of using either GraphHopper edges - which are small, but do not have junctions - or OSM ways, which can be larger but would allow for more usable metrics and would make the system less dependent on GraphHopper.

Ultimately, GraphHopper's edges were chosen as the base unit for cells. This was primarily due to the fact that GraphHopper does not have an internal mapping from its edges to OSM Ways, and does not return which Ways are used as part of a routing response. Additionally, GraphHopper provides an `AllEdgesIterator` that returns both the maximum edge ID as well as the data for every edge in the graph.

The next decision was how and where to store the cells. In the original paper, we saw that each vehicle was represented by its velocity stored as an integer in the cell array. As we are storing a much larger amount of information about each vehicle in the Vehicle class, this is not a viable option. If vehicles store their own velocity and keep track of the current edge and cell they are on, the cells only need to know if any vehicle is present in a cell. As such, each cell is simply a boolean value in an array. The cell is set to `true` if a vehicle is in the cell and `false` if the cell is empty.

In terms of physical storage, the are a number of options. At its core, we need a mapping from edge IDs to cell arrays. In Java, this would usually be represented as a Map of Lists (`Map<Integer, List<Boolean>>`). However, the edge IDs in GraphHopper have been designed to be sequential starting from 0, with the highest ID accessible from the `AllEdgesIterator`. This means that we can instead use a list of lists (`List<List<Boolean>>`). This has the advantage of allowing the lists to grow in size dynamically, but may have performance issues. In Java, Lists must store other Object types rather than primitive types. As objects in Java are all pointers, each Boolean will require a pointer to a separate memory location that holds the `true` or `false` value. This is likely to harm cache performance, something particularly important given the frequency with which the cells will be accessed.

Instead, we can simply allocate a two-dimensional array of the primative boolean type for each edge as a raw array. This avoids the issues with pointers and is consistent with the way GraphHopper stores edges.

However, there is another challenge with regards to storing the data. When parsing the ways, GraphHopper identifies the direction of the edge - each edge can support moving either forward, reversed, or both. When a route is provided by the routing engine, it is in the form of a list of `EdgeIteratorStates`. The `EdgeIteratorState` class has been designed to hide the 'true' direction of the edge, instead switching which node is the base node and which is the adjacent depending on the direction the vehicle is travelling. The result of this is that every edge received from the engine appears as a forward edge. For the cell model, we need to have two separate directions for the roads, or vehicles going in opposite directions will crash into each other and block the roads.

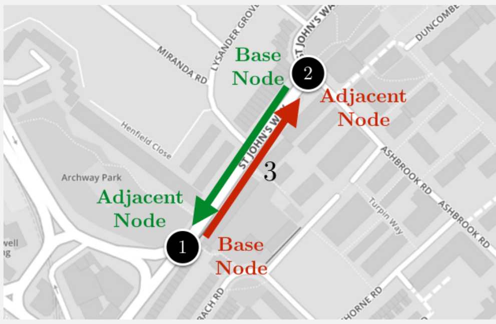


Figure 3.3: The problem with GraphHopper edges

In the diagram above, we see two nodes with ID 1 and 2 joined by an edge with ID 3. Imagine we've performed two routing requests, one in red and one in green. Both routes go through edge 3, but in opposite directions. However, both edges will return true from the `isForward` method of `EdgeIteratorState`, despite going in opposite directions. Although GraphHopper knows that one of these is in reverse, the data is hidden from us.

Figure 3.3 illustrates why this is an issue. We are unable to tell if we are travelling forwards or backwards based on the `EdgeIteratorState` alone, as the ID is the same and the `isForward` method returns true for both cases. This would not make it impossible to identify if the forward or reverse cells should be used for the vehicle. Initially, there was concern that a more complex solution than the boolean arrays would be required - perhaps some kind of 2D mapping from pairs of node IDs (or a base node and edge node) to a cell array. However, it is important to note that we don't need to know if GraphHopper internally stores an edge as forward or reverse so long as we are able to distinguish between the two cases shown in Figure 3.3.

The technique we discovered to solve this uses the fact that the base node and adjacent node returned by the edge changes depending on direction. As such, we can define edges where the base node is greater than the adjacent node as 'forward' and edges where the base node is less than the adjacent node as 'backwards'. This allows us to reliably differentiate between the forward and backwards cases, and save the vehicles from crashing into each other.

The `CellGraph` class handles the storage for the cells, providing convenient getters and setters for edges at specific cells. It also transparently handles the forward and reverse edges, storing two boolean arrays (`boolean[][] cells` and `boolean[][] reverseCells`) and uses whichever one is appropriate for the current EdgeIteratorState.

Cell Size

One important question that must be answered is how many cells should be created for each edge. According to the original paper, each cell should represent the space a single vehicle takes up. In Europe, the average car is around 4.5 metres long. Assuming that cars have at least a one metre gap between them suggests that the cell size should be at least 5.5 metres. However, in the US the average car length is 5 metres, suggesting 6 metres or more would be an appropriate cell size. To accomodate these use cases and others, the size of cells is a configurable parameter, allowing the users to pick an appropriate size for the vehicles they are simulating.

3.3.2 Vehicle and Cell Iterators

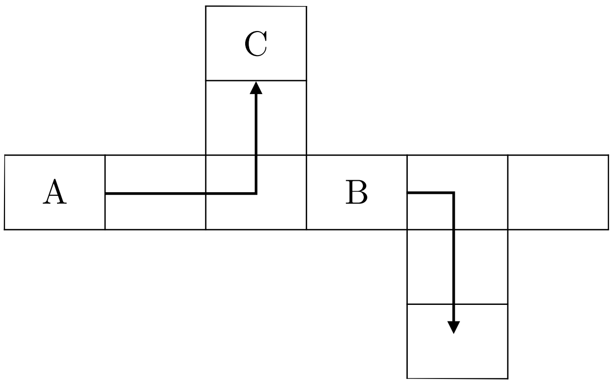


Figure 3.4: The Nagel-Schreckenberg Model across edge boundaries.

As we saw in Section 2.1.3, one of the key parts of the Nagel-Schreckenberg model requires vehicles to know how far they are from the next vehicle. In 3.4, we see three vehicles - A, B, and C. Vehicle A is 2 cells from B and 3 cells from C. A is soon to turn left onto the edge with C, meaning it should not consider B as ahead of it.

However, each of these paths consists of multiple edges. We need a way of treating lists of edges as a single continuous array of cells. This is a good use case for the iterator pattern, providing us with an abstraction over the underlying arrays.

The `VehicleIterator` class is responsible for storing and moving through the vehicle's route. It stores the list of `EdgeIteratorState` objects and moves to the next element each time its `next` method is called. It implements the `EdgeIteratorState` API for full compatibility with GraphHopper. In addition to the usual `EdgeIteratorState` methods, it can return the road speed of an edge. It also removes the first and last elements from the list of edges, as these are virtual edges created by GraphHopper that do not exist in the `CellGraph`.

The `CellIterator` class stores the cell ID and the `VehicleIterator`, meaning it can traverse across edge boundaries by calling the `VehicleIterator`'s `next` method. It also has a reference to the `CellGraph` so that it can return whether there is a vehicle in the current cell.

3.3.3 Vehicle Class

The `Vehicle` class now has five core methods that are used for running the simulation, rather than the single `calculateStep` function used before. The `accelerationStep`, `slowStep`, `randomStep` and `moveStep` all correspond to the four steps of the Nagel-Schreckenberg algorithm, whilst the `updateLocation` method interpolates the cell location into the edge geometry to find the vehicle's location.

The class also stores two core things that make the implementation possible. The first is a `VehicleIterator` named `route` containing the list of edges each vehicle will take, as well as its current edge. The second is the current cell ID the vehicle is on. By duplicating the `VehicleIterator` and passing it to a new `CellIterator`, the vehicle can find out if it is able to accelerate or must slow down - even if the vehicle in front is multiple edges ahead.

```
1  public void slowStep()
2  {
3      int j = 0;
4      CellIterator c = new CellIterator(new VehicleIterator(route), cg, cellId);
5
6      while (!c.next() && j <= v)
7          j++;
8
9      if (j <= v)
10         v = j;
11 }
```

Listing 3.1: The `slowStep` implementation making use of the CellIterator

Listing 3.1 shows how the CellIterator and VehicleIterator work in conjunction to implement the slow step of the Nagel-Schreckenberg method. The variable `j` represents the distance to the next vehicle, whilst `v` is the vehicle's current velocity. Line 4 shows the initialisation of the CellIterator - note that `route` is a VehicleIterator being created with a copy constructor, meaning that the current location of the vehicle will not be changed from this CellIterator.

3.3.4 Results and Evaluation

At this point, the system was capable of simulating any number of vehicles with realistic traffic flow behaviour. However, there were still many key performance improvements that needed to occur before the system would be ready for algorithm development.

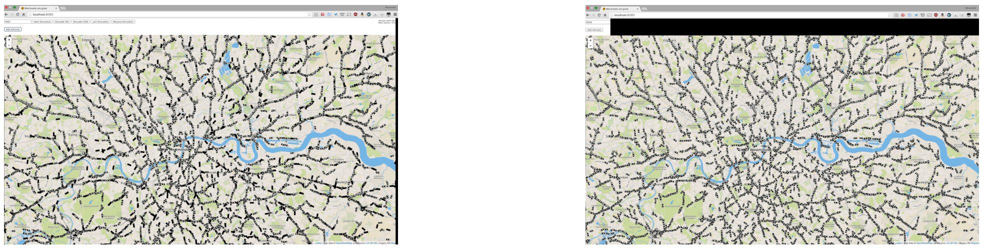
- The back end still waits one second per iteration.
- The back end architecture does not allow for vehicles with different behaviour.
- The back end does not make optimal use of the multiple threads and cores that modern computers have available to them.
- The front end has graphical glitches and poor performance when showing more than 16,000 vehicles.
- Metrics for later data analysis are not recorded.
- Additional vehicles can not be added to the simulation once it has started.
- The simulation cannot be paused or resumed.

3.4 Performance and Architecture Improvements

There were two core goals for the system at this stage. Firstly, improve performance such that at least 64,000 vehicles can be simulated and secondly, modify the architecture of the system to support multiple vehicle types with the ability to have different vehicle types running side by side.

3.4.1 Front-end Performance

Before this stage of the project, the front end had many performance issues. With enough vehicles, the entire page would become slow to respond and update. Even with the one second delay, it would sometimes fail to update before receiving the next set of data.



(a) Vehicles showing up as black squares rather than image icons.

(b) Large black blocks showing over parts of the screen.

Figure 3.5: Graphical glitches caused by displaying more than 16,000 images.

Marker Rendering

Initially, the Leaflet.AnimatedMarker library was used to make it appear as though the vehicles were driving around the map in real-time. However, this had a huge performance cost, and as such had to be disabled.

Additionally, the vehicle image used was reasonably large and in colour. Creating a much smaller vehicle image (from 15KB to 1KB) improved performance a little, but did not raise the 16,000 vehicle limitation.

One of the core issues with the existing implementation was the use of img tags for displaying the vehicles. As each HTML tag is a full blown element in the DOM, there is a reasonably high performance cost associated with their use. There are two main alternatives for showing graphics in the browser - SVG and the HTML5 Canvas API.

Figure 3.6: SVG Rendered orange markers showing 30,000 vehicles

Unlike the img tag, the Canvas API is designed for drawing and other graphical purposes. Instead of one tag per vehicle, A single <canvas> element would be able to render every vehicle. We had been concerned that there would be major challenges in extending the Leaflet Marker class to use an entirely different way of showing markers, but thankfully Leaflet provided the location to display the markers in pixels to Marker subclasses. Using Canvas offered drastically improved performance with seemingly no upper limit on how many vehicles the front end could display with no glitches.

However, there were a few downsides to using Canvas. Firstly, the map correctly re-rendered all the markers when zooming in and out, but did not update the markers when panning the map. Attempts at fixing this issue were unsuccessful. Additionally, the img tags were able to support a popup when mousing over a vehicle that showed its current speed and ID, which was useful for debugging. This would have to be implemented by hand to be supported with Canvas - manually detecting the current mouse position, checking which vehicle it was hovering over and then drawing a custom popup on top of the view.

Instead, we experimented with SVG, an XML based web API for vector graphics. One key advantage is that Leaflet has built in support for drawing certain types of SVG elements. The static image marker could simply be replaced with a `CircleMarker`, which used the same API as the Marker and AnimatedMarker, but instead shows a circle with customisable radius and colour.

SVG combined the performance of Canvas (rendering 64,000 vehicles whilst remaining responsive) and the interactivity of img tags (panning around the map worked flawlessly, and popups worked with no additional work). The only downside was that a circle is shown instead of a vehicle icon, meaning it is not possible to rotate the shape to indicate which direction the vehicles are travelling. This is a minor price to pay for the performance improvements, so the decision was made to use SVG for the front end moving forwards - the result can be seen in Figure 3.6.

WebSocket Data Format

The back end initially just sent down data in string format - for example a vehicle with ID 10067 could have a string of *10067|51.5306611887|0.1609468558|1*. This format has a number of issuses. Firstly, there is limited precision for the values, as they must be truncated to a certain number of decimal places. Secondly, the format is text based, meaning there is a much larger overhead of space compared to storing the values numerically. Finally, it takes additional time to convert the data into a string only to parse it back into numerical values on both the front end and back end.

The string data required for updating the locations of 1000 vehicles takes 33Kb. Encoding this as raw binary data requires 4 bytes for the vehicle ID and speed each and 8 bytes for each of the latitude and longitude, requiring 24 bytes per vehicle and hence 24Kb for the total transfer. This is not only smaller, but also comes with a performance and accuracy improvement as well.

Thankfully, the WebSocket protocol supports the sending of binary data, which can then be efficiently processed by the front end using the DataView APIs. On the back end, we create a single reusable BinaryBuffer object and pass it to the Vehicles, which add their data into a specific offset. This is significantly more efficient than the string based method, although the change is primarily for improving network and front end performance.

3.4.2 Back end Performance

Removing the time-lock

The most necessary change was removing the added delay for the timestep from the Marmoset class. Instead of sending the data every second, the socket server waits for a 'next' message from the front end to start processing the next timestep. It then sends the data and continues to wait for the command to continue.

This drastically improved the speed of the system. For low numbers of vehicles, each timestep takes little more than a few milliseconds on both the front and back end. 1000 iterations with 1000 vehicles used to take at least 16 minutes - after making this change it took just 50 seconds.

Additionally, this allowed much larger numbers of vehicles to be simulated and visualised at the same time - the back end will only send data to the front end once it has rendered the previous set of changes.

This also made it easier to add the ability to pause the simulation - by simply ceasing to send the `next` message, the back end does no further work, leaving the user able to explore the current state of the simulation in the browser.

Exploiting Parallelism

Initial versions of the system simply did all their work on a single thread. There were a number of places were task-level parallelism made it easy to add more threads and cores to the system to improve performance.

```java
public void timestep()
{
    vehicles.stream().forEach(Vehicle::accelerationStep);
    vehicles.stream().forEach(Vehicle::slowStep);
    vehicles.stream().forEach(Vehicle::randomStep);
    vehicles.stream().forEach(Vehicle::moveStep);
    vehicles.stream().forEach(Vehicle::updateLocation);

    vehicles.stream().filter(v -> !v.isFinished()).collect(Collectors.toList());
}
```

Listing 3.2: The single-threaded `timestep` function

In Listing 3.2 we can see the use of the Java 8 Stream API to run the Nagel-Schreckenberg functions on each vehicle. One of the core features of the API is built-in support for concurrency. By replacing the calls to `stream()` with `parallelStream()`, Java will automatically split the computation over multiple threads, whilst ensuring that all processes have completed when the next line is executed. The `vehicles` list was replaced with a synchronised list to ensure that removing finished vehicles from the collection was still thread-safe.

When adding new vehicles to the simulation, their initialisation uses GraphHopper to find the route to their destination. This takes orders of magnitude longer than the timesteps themselves, and as GraphHopper routing requests are thread-safe is an easy way to further improve performance.

3.4.3 Architecture Improvements

3.4

One of the issues with the system at this point was the lack of support for multiple vehicle types - although there is a `Vehicle` class that could have been be subclassed, it was not easy to modify its behaviour without having to rewrite large portions of the code. As such, this part of the system was refactored to improve the architecture of the system.

Vehicle Refactoring

Following the refactor, the `Vehicle` and `VehicleIterator` are now interfaces of the public methods originally implemented in the Vehicle class. `BaseVehicle` and `BaseVehicleIterator` are abstract classes that implement most of the boilerplate functionality. `BaseVehicle` holds the implementation of the Nagel-Schreckenberg model, with routing behaviour defined by the `VehicleIterator`. A subclass of `BaseVehicle` must implement only a single method - `getVehicleIterator()`, which creates a `VehicleIterator` of the correct type and with the right routing information.

For the standard routing algorithm, new `DijkstraVehicle` and `DijkstraVehicleIterator` classes have been created. As a way of verifying that multiple vehicle types are possible, a `RandomVehicle` and corresponding `RandomVehicleIterator` that simply drive around at random with no overarching behaviour were implemented.

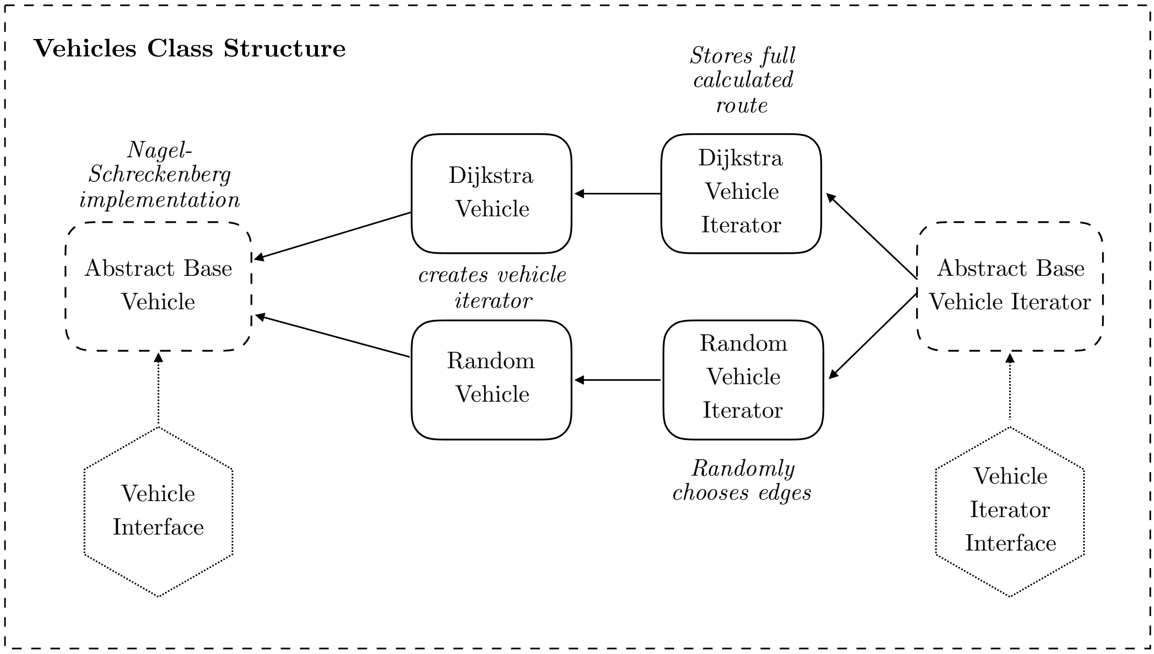


Figure 3.7: The Vehicle and VehicleIterator class structure

3.4.4 Metric calculations

The initial engine only allowed for viewing the results of an algorithm by looking at the visualisation - it is also important to be able to perform offline analysis of the data.

Brief calculations showed that storing the full movement of 64,000 vehicles could take up to 10GB of storage, suggesting that it was better to calculate metrics at each timestep and store those instead. There are three main types of metric - vehicle, route and network metrics.

Vehicle Metrics

As their name suggests, vehicle metrics provide information about the state of one or more vehicles. They can be calculated at any time, so can be recorded and stored on each iteration. Vehicle metrics are the easiest way of getting a sense of the overall behaviour of the system at a given point in time.

With the goal of understanding the levels of congestion, two main vehicle metrics were tracked: the number of vehicles not moving at the maximum road speed and the number of vehicles that slowed down due to a vehicle in front of them.

A new inner `Metric` class was added to `MarmosetHopper`, with simple data properties storing the average values for the two metrics mentioned above, as well as the total number of vehicles and the average speed (in cells per second) of the vehicles. The `toString` method has also been overridden to print the data in CSV format, with a separate static method to return the header for the CSV file.

Route Metrics

We can also better understand congestion by looking at the length of time it takes for a vehicle to reach its destination. Inside the simulation engine, the `Vehicle` class simply tracks the number of iterations it has travelled before reaching its destination.

The GraphHopper `GHResponse` class stores the length of time a route is expected to take, which can be compared to the actual journey time. Although it does not include a realistic model of vehicle behaviour (so does not truly represent the correct time a journey would take, even on empty roads), the difference between expected and actual time is another metric that can be used to compare different vehicle behaviours.

Route metrics can only be recorded once the vehicle has reached its destination rather than whilst it's travelling. As such, a `printMetrics` function was added to `BaseVehicle` with an implementation in `DijkstraVehicle` that prints the expected time from GraphHopper and the actual number of iterations the vehicle took to reach its destination.

These metrics are output into a file named after the ID of the vehicle in question - each vehicle has its own file, meaning there is no need for centralised management of writing to disk.

Network Metrics

Network metrics provide information about the roads themselves and how congested or occupied they are. Instead of providing numerical data for every edge (which provides very little information given how few cells/vehicles the average edge has), a visual approach was used, simply retaining the full set of vehicle positions and speed every 1000 iterations.

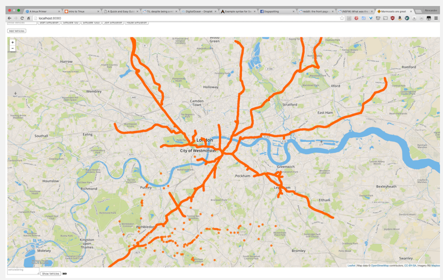


Figure 3.8: Post-simulation analysis of the 9,000th iteration of 15,000 vehicles

The code previously used to convert the Vehicle objects into strings was re-purposed, with the output placed into an iteration file. This means that the data is human readable and easy to parse. On the front end, an input box with basic parsing code was added to allow the data to be explored interactively after the simulation has finished, as can be seen in Figure 3.8.

3.4.5 Event System

Although the refactors mentioned above do allow for a large amount of flexibility, using new vehicle classes (or adding new behaviour on each timestep) required modifying the `MarmosetHopper` class directly. Using an events system enables users of the engine to add code and functionality during key points of the program's execution without directly modifying the engine. Instead, users can call a `listenTo` method with a callback function that is executed when the event is triggered.

Event triggers were added in key locations so that these features would be useful. For example, an event is triggered at the start and end of each timestep, at the beginning and end of initialisation and when a new vehicle is added.

3.4.6 Offline Simulation

With the addition of metrics, it became clear that it would be useful to be able to run a simulation in the background and analyse the data later. One use case would be understanding how long it takes all vehicles to reach their destinations; another is for analysing very high density networks where it may not be desirable to view the data on screen in real-time due to the added performance cost.

A command line flag (`--file`) was added with an argument for the number of vehicles to be simulated as well as a custom name. The simulation creates a folder for the metrics including the number of vehicles, a Unix timestamp, and the chosen name, allowing it to be uniquely identified.

However, the offline simulation appeared to be running slower than hoped given it was not constrained by the browser rendering. The VisualVM Java profiler was used to identify potential performance improvements whilst simulating. Far more time was spent in the `updateLocation` step than expected - and given that the physical location is only used for display and not internal routing, a function parameter was added to disable the location updates when simulating offline. Instead, the location is only updated when required (e.g for outputting the locations every 1000 iterations).

Hot Spots – Method	Self Time [%]	Self Time	Self Time (CPU)	Total Time	Total Time (CPU)
com.graphhopper.marmoset.vehicle.BaseVehicle.**updateLocation** ()		2,196 ms (24.7%)	2,196 ms	2,392 ms	2,392 ms
com.graphhopper.marmoset.vehicle.BaseVehicle.**addToBuffer** ()		603 ms (6.8%)	603 ms	703 ms	703 ms
com.graphhopper.marmoset.vehicle.DijkstraVehicleIterator.**duplicate** ()		279 ms (3.2%)	279 ms	279 ms	279 ms
com.graphhopper.marmoset.MarmosetHopper$$Lambda$15.1276042862.ac		102 ms (1.2%)	102 ms	2,494 ms	2,494 ms
com.graphhopper.marmoset.MarmosetHopper.**timestep** ()		93.3 ms (1.1%)	0.000 ms	2,587 ms	2,494 ms

Figure 3.9: VisualVM Profile of the most time consuming functions

Removing the location updates had a drastic improvement to offline performance, reducing a simulation from over 3 hours to under 7 minutes. Simulations with large numbers of vehicles could now be simulated to completion in a reasonably short amount of time.

3.5 Algorithm Design and Implementation

As a way of exploring the capability of the simulation engine, a multi-vehicle routing algorithm was designed, implemented and tested.

3.5.1 Algorithm Design

We chose to design a V2I algorithm, as there would be a substantial amount of additional code required to simulate realistic V2V communication. It is also beneficial that V2I simulation can apply to both high and low density situations as it is not limited by the distance to the nearest vehicle. Furthermore, V2I algorithms that rely only on mobile connections are able to work with both human drivers and self-driving cars, unlike V2V algorithms that would require all vehicles to have new hardware and software. Instead, access to a central system that knew the location and planned destination of every vehicle in a city is assumed. The algorithm then provides routes for each vehicle in the system.

The initial design of the Marmoset algorithm is described below. It is important to note that this is not the final algorithm - the goal was to find a technique that could potentially be effective and then optimise, improve and understand it using the simulation engine.

Vehicle Behaviour

Individual vehicles make their current route and position available to the central system at all times. The vehicles request routes from the system and then follow them directly. A vehicle requests a new route after a random interval (based on a predetermined parameter).

Central System Behaviour

The central system stores the number of vehicles that have been told to route along each edge, called the Expected Map. It is initialised with zero values for all edges on start-up.

At a fixed interval, the system updates the Expected Map by multiplying every edge by a damping factor that is less than 1. It then adds the edges in each vehicle's current route to the Expected Map.

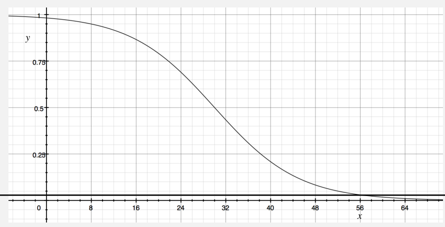


Figure 3.10: The initial choice of congestion function.

When a route is requested, the data from the Expected Map is used in conjunction with a congestion function to modify the weight of edges based on their expected density. This will return a route that avoids high-congestion roads, which will be reflected in the Expected Map upon its next update.

The graphs in Figures 2.3a and 2.3b were similar in nature to a sigmoid function such as tanh, so this was used as the foundation for the congestion function. Adjusting the scale and position of the function led to the graph shown in Figure 3.10, $v(x) = \dfrac{\tanh(2 - \frac{x}{15}) + 1}{2}$.

Design Justification

The goal of the Expected Map is to help vehicles avoid roads that are likely to have high levels of traffic. However, a naïve approach to this would result in oscillatory behaviour between optimal and sub-optimal routes, as can be seen in Figure 3.11. As such, the Expected Map is retained for further requests. The first step of updating the Expected Map multiplies each value by the damping factor. This is to stop the system from having too much of a memory (and hence over-penalising roads that will have many vehicles on them). This factor can be adjusted to define how much memory the system will have of the vehicles' routes.

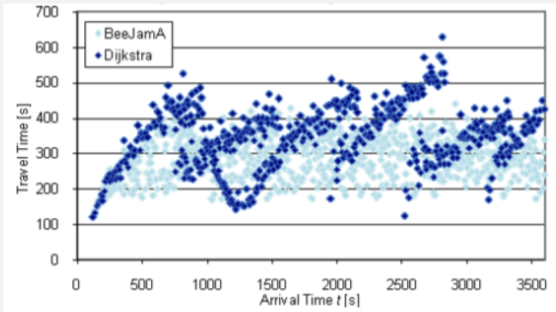


Figure 3.11: Oscillation of travel time when using Dijkstra's algorithm with traffic information.

Furthermore, the system can handle the expectation that most routes will change at some point whilst the vehicle is on the road. The random rerouting was chosen to ensure that the busy areas will not be avoided by all vehicles once the Expected Map has shown that they will be busy. If every vehicle was rerouted simultaneously, the vehicles would completely avoid the busy roads and hence fail to use the network capacity available to them.

3.5.2 Implementation

The implementation of this algorithm made use of the new event system and required the subclassing of two GraphHopper types. A central self-driving vehicle controller class was also created that manages the behaviour of all the vehicles.

In keeping with the new system architecture, `SelfDrivingVehicle` and `SelfDrivingVehicleIterator` classes were created. The SelfDrivingVehicle class is similar to the DijkstraVehicle, but offers new methods for recalculating the route using the expected weighting. The SelfDrivingVehicleIterator is a subclass of the DijkstraVehicleIterator, but adds a simple method for replacing the list of edges once a new list has been calculated by the SelfDrivingVehicle class.

The Expected Map was implemented by creating a GraphHopper weighting class, `ExpectedWeighting` (see Section 2.2.2). It stores an array of floating point values representing the Expected Map (`double expectedMap[]`) and has an `updateExpectedMap` method that applies the damping factor and adds vehicle routes to the map. The weighting's `calcWeight` method uses the curve described in the previous section to return a weighting for an edge given the value in the Expected Map and the speed of the road.

This is all orchestrated by the `MultiSDVController` class, which stores a list of all `SelfDrivingVehicles`. It subscribes to the "vehicle:add" event to create its own list of vehicles, as well as the "timestep:end" event for triggering the re-routing and updating of the Expected Map.

The `MultiSDVController` was also optimised to exploit parallelism for the re-routing operations. As it was not a use case suited to the parallel stream API, an `ExecutorService` is used to manage and shutdown the threads for routing as required.

3.5.3 Testing

Algorithmic Adjustments

Initial tests of the algorithm showed poor performance and many failed routes. Use of the visualisation aspect of the routing engine was crucial in identifying the causes of this.

The first issue was caused by the congestion function. As the expected value for each edge represents how many vehicles will ever drive on that road (rather than at one time), the numbers are usually much larger than the number of vehicles that would be on the road at any given time. As such, the values returned by the congestion function were often very close to (or sometimes exactly) zero. The end result of this was that after a few recalculations of the Expected Map, any popular edge was completly removed from the graph, and hence connections to high-congestion areas of the map were removed entirely.

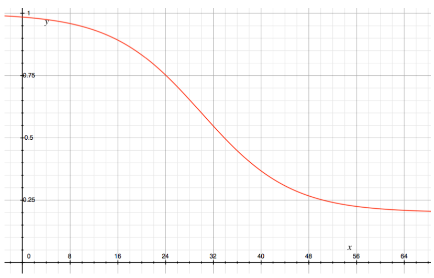
It is not entirely undesirable to remove roads high congestion roads from the graph. However, this would require perfect knowledge that a given road will be blocked - due to the non-deterministic nature of the vehicles (caused by the re-routing opreation), this cannot be guaranteed and hence edge removal is not appropriate for the routing algorithm. In response to this discovery, the congestion function was modified to have a lower bound of 25% of the original road speed, using the same curve shape as before. The function used can be seen in Figure 3.12a.

The other issue was that many roads near a vehicle's destination would have very high values in the Expected Map, creating routes that avoided moving towards the destination. This was partly due to the fact that the vehicle was attempting to avoid road penalties created by its own routes. This was fixed by using a progress function - instead of simply incrementing the value at the edge by one, it is incremented by a value determined from the progress function. The progress function takes in a percentage - the edges percentage in the route - and outputs a value that decreases as we reach the vehicle's destination. The justification for this behaviour is that a vehicle is much more likely to visit an edge nearer to it than further away, as it has a higher probability of having re-routed by then. The progress function can be seen in Figure 3.12b.

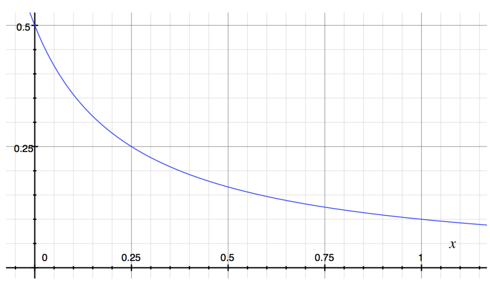
Additionally, it was found that random routing did not have the desired effect - many vehicles never adjusted their routes, whilst others re-routed as many as seven times during a single simulation. This was problematic as the end result was too many vehicles remaining on the highly congested routes, even if the Expected Map showed that they should be avoided.

Parameter Adjustments

The algorithm offers five core parameters that can be adjusted, listed below.



(a) The congestion function.
$$v(x) = \frac{\tanh(2 - \frac{x}{15}) + 1.5}{2.5}$$



(b) The progress function.
$$f(x) = \frac{1}{8x + 2}$$

- **Congestion function** - can be modified to reduce penalties overall, behave differently for different types of road, or have a longer or shorter tail.
- **Progress function** - can be modified to decrease or increase values on nodes further away.
- **Re-routing percentage** - can be modified to change how quickly the vehicles react to high-congestion areas.
- **Damping factor** - can be modified to change the memory of the Expected Map.
- **Expected map update frequency** - can be modified to change how fast the system reacts to vehicles changing their routes.

After a few minor experiments, the congestion function shown in Figure 3.12a was used for testing the algorithm, as it penalised busy roads without causing overly long routes for most vehicles. Likewise, the progress function in Figure 3.12b was chosen to ensure that central roads would not be repeatedly penalised and cause undesirable results.

The damping factor was initially chosen as 0.6, meaning that the edges kept 60% of their value when the Expected Map was updated. This proved to be too high, causing extended travel times when simulating. After brief experimentation, a lower value of 0.2 was used and found to be more effective.

The re-route percentage has an impact on both the behaviour of the vehicles and the performance of the engine - the more vehicles it has to reroute on each iteration, the longer the simulation takes overall. With a re-route percentage of 0.1%, simulating 64,000 vehicles requires 640 re-routing operations at each timestep. On a many-core server, this took approximately one second per iteration. This was a slow but acceptable compromise, as the simulation could be left to run overnight and analysed after the fact using the metric data. Furthermore, the speed of each iteration increases as more vehicles reach their destination, due to the number of vehicles to re-reoute decreasing.

There are also a number of parameters that must be selected for the simulation engine itself.

- **Cell size** - can increase or decrease number of vehicles each road can have, also affects max speed of roads.
- **Slow probability** - modifying the probablity a vehicle slows down (as described by the Nagel-Schreckenberg algorithm) can be adjusted to simulated different types of driving behaviour.
- **Vehicle destinations** - start and end location of vehicles can be adjusted to better model/understand traffic flow under different conditions.

Due to the increasing size of European vehicles and to account for larger vehicles like trucks or vans, 6.5 metres was chosen to be the size of cells in simulation. Brief tests showed that this had a negligible impact on the overall performance of the algorthm, having the same effect as minorly reducing the vehicle density.

For the slow probability, a value of 0.4 was chosen based on research in [24]. There are various valid values for the probability, ranging from 0 to around 0.5. If the value is too low, the model does not simulate traffic jams, whilst a value too high would fail to match reality.

A simple model of rush hour was created for the purpose of evaluating the algorithms. Each vehicle is given a random location and destination within London. The starting location is randomly distributed within the entirely of London, appproximately bounded by the M25. The destination is randomly distributed across the centre of London only, roughly covering Zone 1. Each algorithm was simulated with 10,000 to 80,000 vehicles in increments of 10,000, with the goal of understanding how traffic flows differently within each algorithm.

Evaluation

This section will discuss testing methodology, with the results of the algorithm and the overall performance of simulation engine discussed in Chapter 4 below.

Evaluating how two different routing algorithms would behave in any situation is a challenging task with a near unlimited scope, particularly with complex algorithms relying on the interactions between tens or even hundreds of thousands of separate entities. As such, the scope of the evaluation has been limited to one key scenario in a particular location. London has been chosen for the location, due to the author's familiarity with its roads and expected congestion. The tests will be a simple simulation of rush hour, when 64,000 vehicles [6] travel into the city centre. Congestion during this period of time is often at its worse, in spite of attempts by TfL to reduce travel at peak times.

Tests will focus on a single parameter at a time. The tests for different cell sizes and vehicle density will be run with both the Marmoset and Dijkstra algorithms, whilst the tests for the Marmoset progress and congenstion functions will only be compared to other Marmoset tests.

Interestingly, when testing the rush hour simulation using Dijkstra's algorithm with 30,000 vehicles or more, unexpected behaviour was discovered. Namely, the simulation would not terminate due to congestion in a closed loop around Elephant & Castle roundabout. All vehicles would slow to a halt along a fixed number of fully congested roads, as can be seen in Figure 4.3.

As such, new termination conditions were added. If every vehicle has slowed due to a vehicle ahead, is not at the road's maximum speed, and the average speed of the vehicles is near zero, the simulation terminates early. For 60,000 vehicles, early terminaton happens after approximately 21,000 iterations - with 30,000 vehicles this state is reached in around 12,000 iterations, whilst 20,000 vehicles terminates completely (with 0 vehicles yet to reach their destination) in a little over 9,000 iterations.

Chapter 4

Critical Evaluation

The goal of this section is providing an empirical and critical analysis of the Marmoset algorith compared to Dijkstra's algorithm, as well as an evaluation of the suitability of the simulation engine for the purpose of multi-vehicle routing algorithm design.

4.1 Algorithmic Evaluation

4.1.1 Dijkstra's Algorithm

We start by analysing the results of simulation when using Dijkstra's algorithm to route the vehicles. Unlike the Marmoset algorithm, the vehicles do not avoid congestion, attempting to find the fastest route to their destination based only on the speed limit of each road.

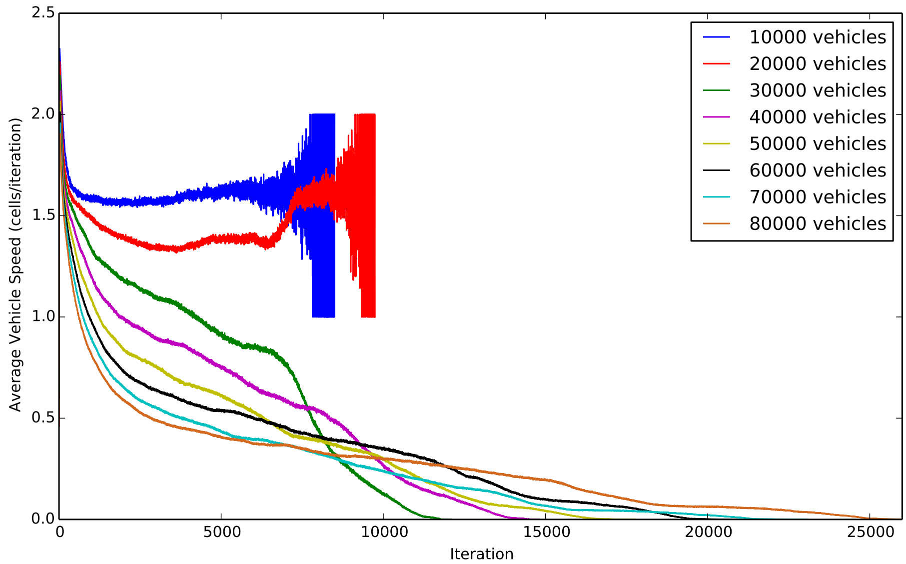


Figure 4.1: Average speed at each iteration for vehicles routing with Dijkstra's Algorithm

The graph in Figure 4.1 shows two different outcomes when routing with Dijkstra's Algorithm. When simulating with 10,000 and 20,000 vehicles, we see the behaviour of the algorithm when all vehicles reach their destination and the simulation terminates successfully. The remaining six simulations show the behaviour of the algorithm when it fails to terminate and the vehicles slow to a halt.

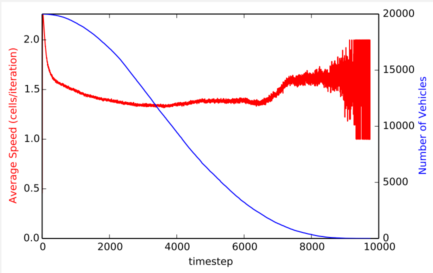


Figure 4.2: Comparison of speed and remaining number of vehicles using Dijkstra's algorithm with 20,000 vehicles.

In the successful termination cases, we see three stages. The average speed starts high, followed by a decline as we enter the congested state. This is followed by a longer period of time where the speed remains mostly stable as vehicles move towards their destinations. Finally, we see a period with increased speed volatility. This is due to the lower volume of vehicles causing the average speed to change much faster. The speed is measured in cells per iteration, which will always be an integer, so the speed of the final few vehicles will oscillate between integer values, as can be seen in Figure 4.2.

In the case of simulations that do not terminate normally, we see the same first two stages (pre-congestion and congestion), but at a certain inflection point the average speed declines faster until it reaches zero. This is most notable for the 30,000 vehicle simulation, where the inflection point can be seen near iteration 7,000, after which the average speed enters a steep decline.

The cause of the simulation failure can be seen in Figure 4.3, where the Elephant & Castle roundabout in central London becomes fully congested, stopping any vehicle from being able to route past it. As the queue of cars continues to grow, more and more main roads become full until no remaining vehicles are capable of movement.

As we can see, although Dijkstra's algorithm provides the optimal routes for the road network, it is incapable of handling the practical concerns that arise when dealing with high congestion environments.

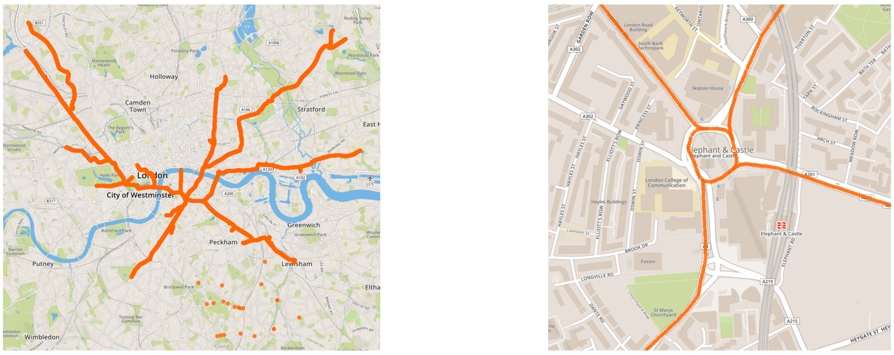


Figure 4.3: The system state shortly before termination

4.1.2 Marmoset Algorithm

We now compare these results to the Marmoset routing algorithm. The most notable difference is that the Marmoset algorithm successfully terminates even when simulating 80,000 vehicles entering the city centre, 16,000 more than in rush hour.

Figure 4.4 shows that the Marmoset algorithm exhibits the same behaviour as Dijkstra's Algorithm for 10,000 and 20,000 vehicles, but without the limitation on the number of vehicles that can be simulated. The data shows the same three stage behaviour as before, with the congested stage elongating as the number of vehicles increases. We can also see a large difference in the amount of time it takes for the simulation to terminate, falling into three separate clusters - less than 40,000 vehicles, 50,000 to 70,000 vehicles and 80,000 vehicles. The reason for this is not clear from this graph alone. To gain further insight into why this clustering may be occuring, we can look directly at the state of the vehicles at specific iterations.

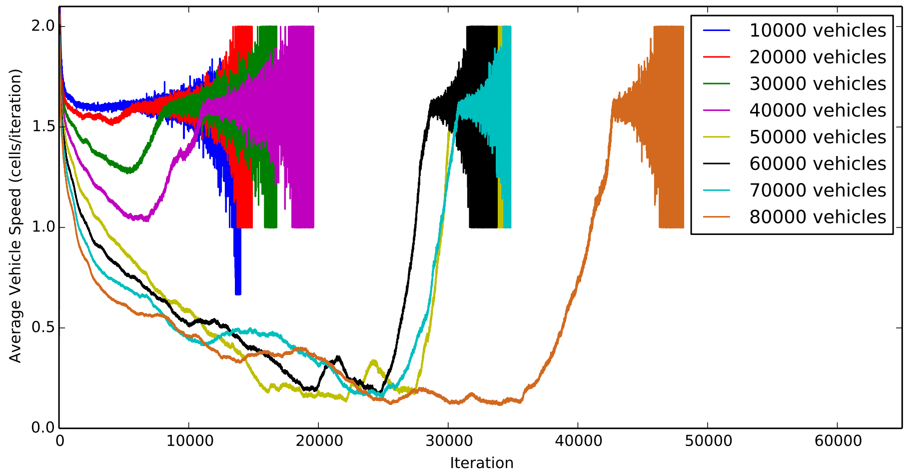


Figure 4.4: Average speed at each iteration for vehicles routing with the Marmoset Algorithm

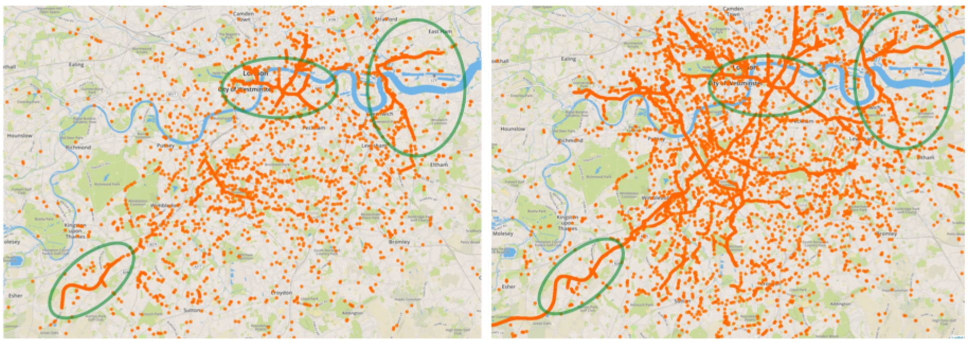
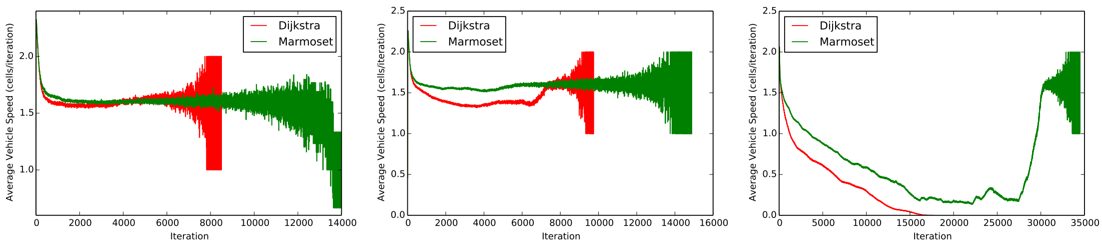


Figure 4.5: The 11,000th iteration of 50,000 and 80,000 vehicles respectively.



(a) 10,000 vehicles (b) 20,000 vehicles (c) 50,000 vehicles

Figure 4.6: Comparison of average speed between Dijkstra's algorithm and the Marmoset algorithm

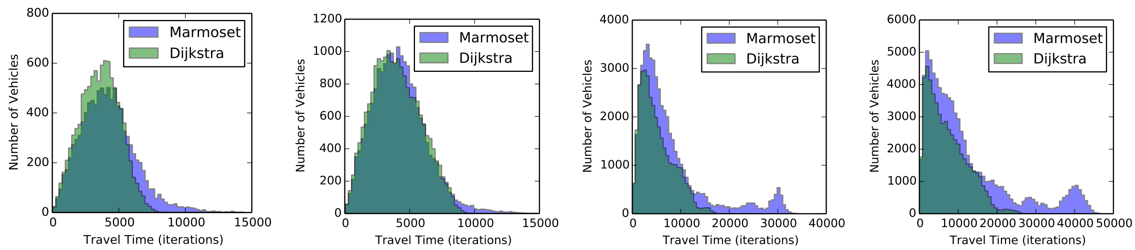
The engine records a snapshot of the position of each vehicle every 1000 iterations. We will compare the state of the system when simulating 50,000 vehicles and 80,000 vehicles at iteration 11,000. This has been chosen as it is approximately close to the inflection point where the vehicles enter their most congested state with the slowest average speed. Note how the 80,000 vehicle iteration takes significantly longer to leave the congested state than the 50,000 vehicle simulation.

Figure 4.5, helps us understand what the cause of this behaviour may be. For the 50,000 vehicle simulation, we see three key areas of congestion, circled in green on both maps. For the lower density simulation, these three areas are separated - once a vehicle has left the congested roads it enters an area of low congestion. In the 80,000 vehicle simulation, this is not the case - instead, the highly congested roads are joined together, requiring significantly longer to clear. We can conclude that the clustering of total travel time is a property of the map and routes that run on it - as the number of vehicles on the road increases, areas prone to congestion increase in size and connect, causing sudden jumps in completion time with only a small increase in the number of vehicles.

Figure 4.6 shows a direct comparision between the two algorithms. We see that when Dijkstra's algorithm does terminate, vehicles reach their destinations earlier than with the Marmoset algorithm, despite a lower average speed. This is likely due to the 'selfish' behaviour of Dijkstra's algorithm, allowing certain vehicles to reach their destinations earlier. With lower vehicles densities, this frees up the roads and enables other vehicles to reach their destinations sooner. However, with more vehicles on the road the congestion blocks vehicles from reaching their destination.

4.1.3 Per-Vehicle Metrics

The data above shows what happens to the vehicles on average, but does not provide us with any further insight on the distribution of travel time or congestion. The vehicle metrics can be used to better understand the impact each algorithm has on travel time for each vehicle.



(a) 10,000 vehicles (b) 20,000 vehicles (c) 50,000 vehicles (d) 80,000 vehicles

Figure 4.7: Histogram comparing travel time of vehicles using each algorithm.

In Figure 4.7, we see some interesting patterns emerge. As demonstrated by the graphs comparing average speed, lower density simulations complete faster using Dijkstra's algorithm. In Figure 4.7a this behaviour is particularly noticable, with the Marmoset algorithm showing a tail of vehicles with long travel times.

Figure 4.7c gives us further insight into the clustering of completion times mentioned above (see Figure 4.4). On the right of the histogram we see a small spike in travel times, centred around iteration 30,000. This supports the theory that congested areas become connected and dependent on each other for vehicles to reach their destinations - once a certain group of vehicles reach their destinations, the group they 'blocked' is able to move once again. The same observation can be made with Figure 4.7d, which has a small spike at iteration 30,000 and another, larger spike around iteration 40,000.

Looking again at Figures 4.7c and 4.7d, it appears that the vehicles routed with Dijkstra's algorithm reach their destinations sooner than those routed by the Marmoset algorithm. This is primarily because the histogram only includes vehicles that have reached their destination, meaning the remaining vehicles never reached their destination at all. We can conclude that this 'greedy' behaviour benefits those that do reach their destination at the cost of the thousands of vehicles incapable of reaching their destinations due to the high levels of congestion.

4.1.4 Vehicle Ratio Analysis

In addition to comparison between the two routing algorithms, we can also analyse their behaviour when driving together on the same roads. As shown in Section 4.1.1, routing using only Dijkstra's algorithm fails to terminate with more than 20,000 vehicles. One of the issues with multi-vehicle routing algorithms is that in order to improve congestion some vehicles have to take longer routes than they would otherwise use. Human drivers would not appreciate the delay and may instead choose to drive on the optimal route. We can analyse the effects of this behaviour by simulating both types of vehicle at the same time and identify what percentage of vehicles 'misbehaving' causes the algorithm to fail.

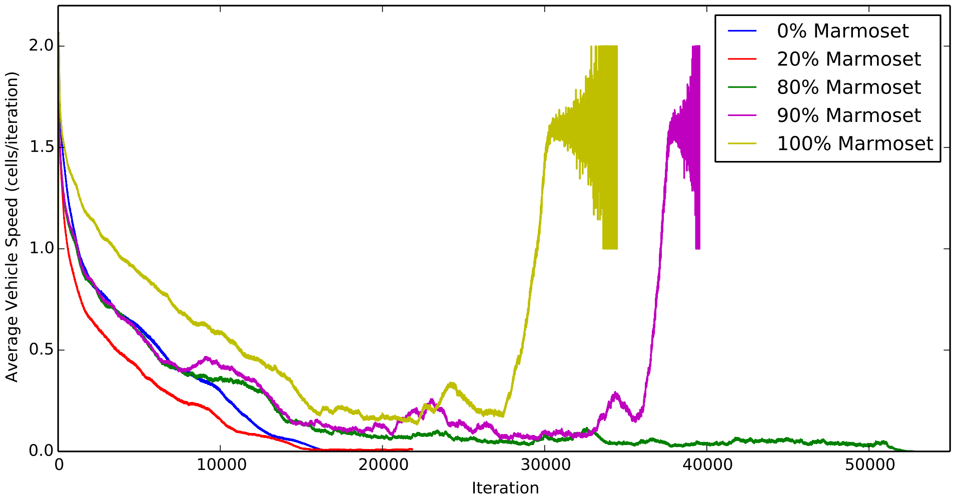


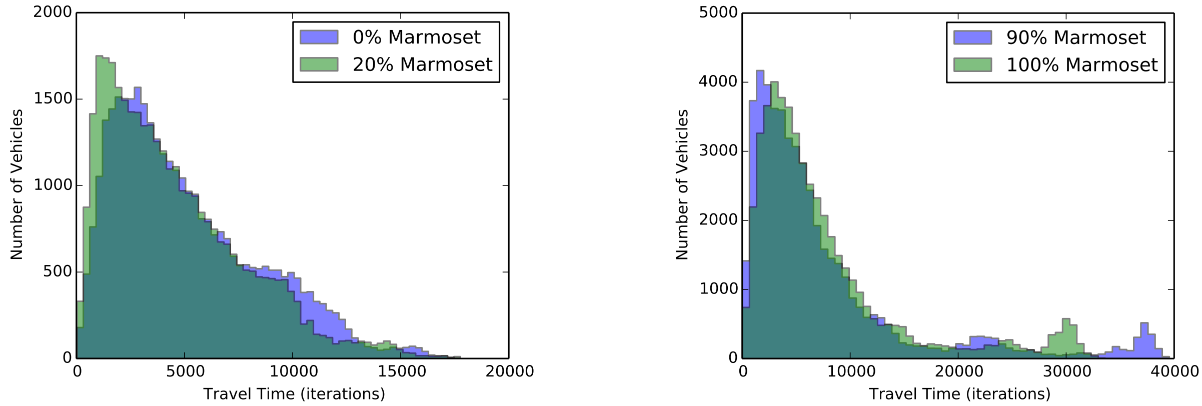
Figure 4.8: Average velocity simulating 50,000 vehicles using the Marmoset and Dijkstra's algorithm.

Simulating 50,000 vehicles in different ratios showed how strong the impact even a small number of non-Marmoset vehicles have on congestion. Surprisingly, even with 80% of the vehicles using the Marmoset algorithm, the simulation failed to terminate. Figure 4.8 shows the average speed of the vehicles when using both algorithms simultaneously in various combinations. Although the vehicles fail to reach their destinations completely with 80% Marmoset vehicles, we do see a change in the behaviour of the algorithm as the proportion of Marmoset vehicles increases. Notably, the time it takes for the average velocity to reach zero is drastically increased - it took 52,801 iterations before the engine decided to terminate the simulation, compared to 16,504 iterations when routing exclusively with Dijkstra's algorithm.

The cause of this is likely the Marmoset vehicles attempting to avoid the central area where the rest of the vehicles are stuck. As the central vehicles are not moving, the routes they have planned will be repeatedly added to the Expected Map, further increasing the penalties for travelling on those roads. After avoiding the central area as much as they can, the Marmoset vehicles eventually join the congested vehicles and the simulation terminates.

We also see changes in both average speed and total completion time when simulating 90% Marmoset vehicles, even though all vehicles reach their destinations. The lack of ability to operate effectively whilst routing with other types of vehicles is a flaw in the algorithm that suggests it may need further optimisation before it is appropriate for real world use - any multi-vehicle routing algorithm would be rolled out slowly rather than all at once, meaning it should be able to handle different types of vehicles on the road. Interestingly, Figure 4.8 shows that average speed with 20% Marmoset vehicles is significantly slower than when routing exclusively with Dijkstra's algorithm.

However, the overall picture does not appear to be so simple. The histogram in Figure 4.9a shows that the vehicles that do reach their destination do so sooner when there are some vehicles using the Marmoset algorithm. One possible cause of this could be that the Marmoset vehicles avoid the higher



(a) (b)

Figure 4.9: Histograms of travel time for different ratios of vehicles.

congestion areas, allowing the Dijkstra vehicles to reach their destinations sooner. This could also explain the lower average speeds, as the Marmoset vehicles would avoid the fastest roads, allowing the nearby Dijkstra vehicles to move towards their destination with less congestion.

In Figure 4.9b, we see the impact on travel time of more vehicles than expected using the most congested routes. Notably, the peak of the 90% histogram occurs earlier than for the 100% histogram, but it has a longer tail with a spike approximately 8,000 iterations later. This is likely due to the Dijkstra vehicles causing a further increase in congestion for busy roads, excaberating the congestion connectivity issue discussed in Section 4.1.2.

4.1.5 Algorithm Performance

The Marmoset algorithm must be fast enough to return routing queries in real-time whilst vehicles are on the road. We will now analytically derive the asymptotic peformance of the algorithm.

We will start by defining the number of vehicles as N and the number of iterations as I (note that this will not be known in advance and could be infinite). Dijkstra's algorithm runs in $O(E \log V)$ when using a binary heap for the priority queue, which is how it is implemented in GraphHopper. The vehicles that route themselves with Dijkstra's algorithm calculate their route once during their first iteration and then iterate through their route until reaching their destination. This means that the first timestep for each vehicle takes $O(E \log V)$ and every subsequent timestep is $O(1)$. Hence for all vehicles, we have a worst case performance of $O(NE \log V))$. For the overall system, we have that as $I \to \infty$, the performance of the simulation overall tends to $O(IN)$. It is also worth noting that the routing step of these vehicles is able to take advantage of Contraction Hierarchies. Analysing the computational complexity of Contraction Hierarchies is a complex task [3], but it is known that in practice their use results in orders of magnitude faster routing [11].

For the Marmoset algorithm, a certain percentage of the vehicles are re-routed every iteration (usually 0.1%). Let this value be c, a constant between 0 and 1. This means that the runtime of each iteration is $O(cNE \log V)$. The rebuilding of the Expected Map takes $O(N)$, which is less than the routing step and hence the runtime of the whole system is $O(cINE \log V)$. Unlike the vehicls routing with Dijkstra's algorithm, we are unable to use Contraction Hierarchies as the algorithm re-weights the edegs on the graph each time the Expected Map is recalculated.

Clearly, the performance of the Marmoset algorithm is worse than routing using Dijkstra's algorithm. This was shown to be true in practice, with the Dijkstra's simulation taking around 15 minutes compared to up to 5 hours for the Marmoset algorithm. However, there are many ways this could be improved in the real world. When simulating, all routing requests must be performed on a single machine - in the real world, the Expected Map could be stored centrally and distributed the vehicles, who then perform their own requests. This reduces the performance of the centralised system to $O(N)$ per iteration, with each vehicle running Dijkstra's algorithm in $O(E \log V)$. As routing is a task many GPS systems are capable of, this seems to be a realistic option for implementing the algorithm in the real world. Furthermore, it is likely that the code and algorithms could be substantially optimised to improve performance were a central system a requirement.

4.1.6 Algorithm Limitations

The approach chosen does have a few limitations. There are a few improvements that could increase the efficiency of the routes generated by the algorithm.

The algorithm's parameters - the progress function, congestion function, damping factor, re-route percentage and update frequency - were chosen based on brief analyses of running a simulation rather than an in-depth exploration on the effect changing each parameter. This would offer further insight into the working of the algorithm as well as improving its performance overall. Unfortunately this was not possible due to the large number of possible combinations of parameters as well as the longer running time for simulations using the Marmoset algorithm.

A key factor that the Expected Map does not take account is the time at which a vehicle will travel on a particular road. If a large number of vehicles were all travelling along the same route but at different times, the Expected Map would predict high levels of congestion and could result in the vehicles taking alternative routes. The progress function was added to help mitigate this, but it does not completely eradicate the issue.

From a more theoretical standpoint, it is clear that the algorithm is not capable of identifying proviably optimal routes in congested situations. In practice, vehicle routing is not deterministic enough for this to be a requirement for a multi-vehicle routing algorithm, but in the interest of evaluating the quality of the routes provided, identifying the best possible completion time for a collection of vehicles would be valuable.

The spikes in the distribution of travel time indicate another area of potential improvement for the algorithm. Ideally, all vehicles should arrive as soon as possible, but in practice this is not necessarily possible. Nonetheless, the algorithm does not incorporate a sense of fairness - some vehicles may have to route much further due to the timing of re-routing operations. Balancing this unfairness may have a detrimental effect on the quality of routes, but an algorithm that felt fair would likely be more popular with drivers.

Finally, the algorithm is not capable of providing an accurate estimation of journey time, as a vehicle's route is likely to change multiple during the journey. Time estimation is an important part of journey planning, so being able to show some kind of expected travel time would be a great improvement.

4.2 Simulation Engine Evaluation

We will now analyse how well the simulation engine has performed in terms of its suitability for algorithm design.

4.2.1 Flexibility

Implementing the Marmoset algorithm required a simulation with the ability to:

- create vehicles with specific behaviour,
- generate routes with a custom function defining the weight of each edge,

- request route data from the vehicles and update the weighting class,
- change the route whilst the vehicles were en route,
- move the vehicles realistically once their route had been planned.

We will now see how the features of the engine made this task relatively easy.

Defining the vehicle specific behavior required creating a new class using the `Vehicle` interface. By simply extending the `DijkstraVehicle` class, the additional functionality was added without difficulty. The `SelfDrivingVehicle` class implements three methods - two for routing and re-routing and one for returning the current path.

Generating routes using a custom weighting was done with the creation of the `ExpectedWeighting` class. This class extended the GraphHopper `FastestWeighting` class, which uses the speed limit and road distance to calculate the fastest possible travel time of a given road. The `ExpectedWeighting` used the `calcWeight` method of the `FastestWeighting`, incorporating its own implementation of the congestion function to return the adjusted weighting.

These two examples show how the use of existing code and powerful abstractions enabled the implementation of new vehicle behaviour without repetitive code or complex integration.

Requesting route data and updating the Expected Map required creating a new class for managing the `SelfDrivingVehicles`. The `MultiSDVController` listened to the "vehicle:add" event and kept track of the vehicle if it was an instance of `SelfDrivingVehicle`. In addition to the vehicles, it stores a reference to the `ExpectedWeighting`. By listening to the "iteration:end" event, the controller can pass information between the `ExpectedWeighting` and the vehicles.

Using the methods defined in the `SelfDrivingVehicle` class, the controller can instruct the vehicles to update and replace their route. The `SelfDrivingVehicleIterator` has a single `resetEdges` method for updating the route.

No additional work is required to create realistic vehicle simulation, as the `BaseVehicle` and `CellIterator` classes handle the implementation of the Nagel-Schreckenberg model.

Together, these examples demonstrate the four key things that have made this fast and adaptable implementation possible.

1. The `BaseVehicle` and `BaseVehicleIterator` abstractions allowed for code with minimal boilerplate or repetition.
2. The event system enabled classes to augment the system with new behaviour.
3. The GraphHopper routing engine allowed the creation of a novel routing strategy.
4. The `CellIterator` and `CellGraph` implementations hid the details of the Nagel-Schreckenberg simulation, working in conjunction with the `VehicleIterator`.

4.2.2 Simulation Engine Limitations

Although the engine has proven effective for algorithm design, there are a number of features that could be added to improve the experience.

With regards to the flexibility mentioned above, one challenge was passing the `ExpectedWeighting` to GraphHopper. This was done by subclassing the `GraphHopper` class into a small `MarmosetGraphHopper` class that overrode the `createWeighting` method, returning the same instance of the `ExpectedWeighting` every time. Although this was effective at solving the problem at hand, it added a fair amount of unnecessary code and is not scalable to additional weighting classes. A better implementation of the `MarmosetGraphHopper` class could allow a weighting class to be added with a single function call.

Currently, the simulation becomes so congested that no vehicles are able to travel when using Dijkstra's algorithm with large numbers of vehicles. Although bad traffic occurs in practice, it usually does not stop the movement of an entire city. Improvements to the Nagel-Schreckenberg model could be made to support multiple lanes [21, 18], likely removing the issue with the Elephant & Castle roundabout. In general, further improvements could be made to add to the realism of the vehicle simulation.

Metrics are an important part of understanding a new algorithm. At the moment, only three vehicle metrics and two route metrics are tracked. Different vehicles and algorithms may want to keep track of certain values, either for vehicles, routes, or the network overall. Adding techniques that allow for this level of customisation without requiring the user to simply write their own metric code would be useful. It could also automatically calculate a number of statistics about each type of data it receives, returning the maximum, miniumum, average and standard deviation of any metric is is passed by default. It could also be beneficial for the engine to automatically generate graphs and histograms when the simulation terminates.

The current performance of the simulation was good, but there is room for improvement. For testing purposes, the engine was run on a 20 CPU server. It was able to complete simulations faster, but only offered a 2-3x performance increase over a consumer laptop, despite having well over 5x the processing power. This could likely be improved through a better multi-threaded architecture and optimisations for the `BaseVehicle` and `CellGraph` classes.

When comparing the ratios of different vehicles in Section 4.1.4, the front end UI does not show any difference between each vehicle type, making a visualisation based analysis impossible. If the UI displayed different coloured markers for each type of vehicle, this could be avoided. In general the front end UI could be improved by displaying additional data, showing the current route a vehicle is planning on taking and showing the current state of the Expected Map.

The engine is currently well equipped to deal with the V2I use case, but there is no built in support for V2V communication. Adding a realistic V2V simulation model and communication protocol would be a major addition to the engine, enabling the simulation and design of many other types of algorithm.

However, in spite of this the engine performs its job effectively. There are cleary improvements that could be made, but it is important to avoid creating an engine like MATSim that is good at many things but great at nothing. The purpose of Marmoset is to be a city-scale macroscopic simulation engine for algorithm design, a job it has proven itself capable of.

Chapter 5

Conclusion

This section will conclude the project, discussing the overall quality of the algorithm and simulation engine before discussing the potential they hold for future research and development.

5.1 Achievements and Contributions

In this section we have revisited the original objectives of the project and discussed how they have been achieved.

1. *Research existing algorithms and simulation tools to identify the strengths and weaknesses of current approaches.*

   Before starting the design or implementation, research was done into the state of the art simulation engines and multi-vehicle routing algorithms.

   An exploration was done into SUMO and MATSim to see if they were appropriate platforms for writing new multi-vehicle routing algorithms. Finding them lacking in either the city-scale or macroscopic requirments of this kind of development, the decision was made to build a specialised simulation engine to aid in the creation of such an algorithm.

   Algorithms for realistic traffic flow simulation were discussed in Section 2.1.3, with the challenges faced implementing them shown in Section 3.3.

   Further research identified a separation between the two different types of multi-vehicle routing algorithm. The first type deals with vehicle to vehicle communications (or a VANET). This can be seen in the Connectivity Aware Routing algorithm [15], creatively known as CAR. The second type deals with vehicle to infrastructure communications, usually with some kind of central management system. The BeeJamA algorithm [23] is an example of a V2I algorithm. Its use of empirically derived congestion functions proved to be a useful insight for the creation of the Marmoset algorithm.

2. *Design a simulation architecture that allows for fast experimentation, easy integration with real world information and full implementation flexibility.*

   The design of the simulation engine was instrumental to its success. In Section 3.4, we saw how the architecture evolved from a simple core into a much improved engine with powerful abstractions and well separated implementations of the traffic flow model. In Section 4.2, evidence for the flexibility of the engine was provided along with the core insights that provide it.

The GraphHopper engine was well equipped for handling the goal of integrating with real world data, providing solid support for OpenStreetMaps. This did not come at the cost of either performance or adaptability, as GraphHopper has been designed to enable full control over the data stored at each edge as well as the class that converts this data into a weighting for the purpose of routing.

The front end visualisation layer made finding insight into the behaviour of an algorithm much easier and faster than using an offline simulation tool such as MATSim. Being able to iterate on the design and experiment with parameters quickly allows for more effective development and ultimately a better result.

3. *Build and optimise the simulation engine on top of existing open source tools.*

   GraphHopper, Leaflet.js, and Java are the three core open source tools that made the simulation engine possible. The technical details regarding the implementation can be read in Section 2.2 and throughout Chapter 3.

   Optimising the simulation engine was an important part of the project. The initial implementation was slow, but with structural improvements to the communication between the front and back end as well as analysis of bottlenecks led to significant improvements in execution speed for both parts of the application. This was discussed further in Section 3.4.2.

   However, whilst the engine itself was fast enough to handle real-time simulation and visualisation of thousands of vehicles, the Marmoset algorithm was not able to take full advantage of this due to the computationally expensive re-routing processs. This meant that beyond the initial choice of parameters it was hard to gain a deeper understanding that changes to the algorithm had on the vehicle behavior and congestion.

4. *Design and implement a novel multi-vehicle routing algorithm.*

   In Section 3.5.1, the design of the Marmoset algorithm was laid out, with the implementation in Section 3.5.2.

   The critical evaluation showed that the algorithm was capable of offering a substantial improvement over using Dijkstra's algorithm, which created blockages stopping large numbers of vehicles from reaching their destination.

   The algorithm uses the idea of congestion functions from the BeeJamA algorithm for vehicle velocity prediction, but has novel ideas in the form of the Expected Map and the progress function.

5. *Use the tools provided by the simulation engine to analyse, understand, and optimise the routing algorithm.*

   In Chapter 4, the performance of the Marmoset algorithm was analysed in depth. Using metrics generated by the simulation engine, we were able to reach an understanding of how the engine behaved with increasing numbers of vehicles and gain an insight into the unexpected clustering of time to completion.

   We were also able to learn about the behaviour of the Marmoset algorithm when routing alongside Dijkstra's algorithm. By showing a histogram of travel times for different ratios of vehicles, we were able to see that despite improvements in average speed, some vehicles had to suffer worsened travel time.

The initial set of objectives in Chapter 1 set out two key deliverables. The first was a simulation engine and the second was an algorithm built on top of it.

The first achievement of this project was building a simulation engine focused on algorithm design that was fast enough to handle the simulation of entire cities. The engine was then used to design, implement and optimise the Marmoset algorithm for multi-vehicle routing.

The architecture of the simulation engine enabled the implementation of a novel and complex algorithm for routing. The four main architectural insights were the separation and abstraction of `Vehicle` and `VehicleIterator` classes, the use of an event system for adding behaviour, the creation and abstraction of the `CellIterator` class, and making use of the flexibility of the GraphHopper routing engine.

The engine was then optimised for speed and ease of use. Running a 64,000 vehicle offline simulation took just 6 minutes 24 seconds or 20.3ms per iteration when using routes provided by Dijkstra's algorithm. A responsive interface for the visualisation of vehicles was created, allowing the users to watch the see the simulation as it is occurring.

Following research into existing algorithms, a novel approach to providing routes for multiple vehicles was designed and implemented. By using insight from the BeeJamA algorithm, a congestion function was identified and optimised, providing a key metric in predicting a vehicle's expected velocity on a given road. The use of visualisation whilst simulating enabled fast iteration and improvement on this function as well as quick insight into the impact it had on congestion overall. In conjunction with post-simulation visualisation, this led to identifying the need for the progress function to avoid over-penalising busy but distant roads.

5.2 Future Work

As shown in Chapter 4, the Marmoset algorithm performed better than Dijkstra's for situations with high levels of congestion. Empirical analysis demonstrated that the algorithm offered an improvement in average speed and avoided mass congestion when dealing with high volumes of vehicles. It was also shown to be able to handle a small number of 'selfish' vehicles driving alonside the Marmoset routed vehicles. In Section 4.1.6, we looked at some of the direct limitations on the algorithm. We will now take a broader look at the potential impact the Marmoset algorithm could have in other fields.

Machine Learning

The congestion and progress functions are suitable candidates for optimisation using machine learning techniques, which could help discover very different functions that an individual researcher may not have considered. This could also be applied to the algorithm's other parameters, finding an optimal configuration between them without requiring an exhaustive search.

Expected Map Intelligence

The Expected Map technique could be extended to store details of roadworks or other planned closures, using the existing process to help vehicle avoid the area and decrease the likelihood of creating additional congestion. Furthermore, data on peak traffic and travel times could be added onto the Expected Map, further anticipating driver behaviour and avoiding certain areas before vehicles are even planning to travel to or through a given location.

Broader Uses

There are many other industries where the reduction of congestion is desirable. Most flights occur on predetermined routes, requiring careful planning to avoid in air collisions and congestion. Likewise, cargo ships and other commercial vessles could benefit from the improvements multi-vehicle routing can provide. With careful adaptation, the Marmoset techniques could be applied to these use cases and many others.

Real World Data Augmentation

At the moment, the simulation uses the mapping data and traffic flow to model vehicle behaviour. Real world data could be collected and used to improve the accuracy of the simulation engine and the Marmoset algorithm. Rather than creating estimates of travel time using only the congestion function, roads could be individually measured to build specialised congestion functions for those specific areas.

Competitive Simulation

The ability to simulate multiple types of vehicle at the same time means we can simulate a likely outcome of capitalism - routing algorithms being sold as a competitive feature, each claiming to be able to get their drivers to their destinations fastest, probably at the cost of other users on the road.

Distributed Computing

Even running the simulation engine on a very powerful computer did not give the performance desired. Making use of either a fleet of vehicles or a cluster of computers would allow for a drastic performance increase. This would require redesigning some major parts of the engine, but due to the existing multi-threaded architecture would allow for reusing many of the core components.

City Planning

The use of OpenStreetMaps allows for easy modification of the input data. This could be used to analyse the effects of building or changing the road network, enabling city planners to make more informed decisions. The use of the Marmoset algorithm would allow cities to be designed with self-driving vehicles in mind rather than just for the current state of transport.

On-Demand Self Driving Vehicles

As discussed in Chapter 1, the popularity of on-demand transport has increased dramatically, with most of the demand going towards Uber. With $9 billion raised from investors, Uber are in a strong position to make long term investments into the future of transport, and have been making substantial invenstments into self-driving vehicles. Their vision is a powerful one - car ownership could be replaced entirely by a massive fleet of self-driving vehicles, picking up and dropping off passengers entirely autonomously.

Understanding how best to design algorithms for this would be essential to finding the best approach to achieving this vision. The behaviour of these vehicles could be implemented as part of a simulation engine to aid in the decision making process.

5.3 Final Remarks

The future of driving and transportation is at a fascinating turning point. For the first time in history, cars will be able to drive themselves, enabling increases in productivity and reducing costs for businesses. For this to be possible, self-driving vehicles will have to be substantially smarter than the human-driven cars that have come before them. Solving this problem requires realistic, powereful simulation tools and new algorithms capable of exploiting the huge quantity of data these vehicles will provide. I believe Marmoset is the first step towards creating this system.

Bibliography

[1] Vladimir Agafonkin, Dave Leaver, et al. *Leaflet.js*. http://leafletjs.com/.

[2] Rutger Claes, Tom Holvoet, and Danny Weyns. A decentralized approach for anticipatory vehicle routing using delegate multiagent systems. *Intelligent Transportation Systems, IEEE Transactions on*, 12(2):364–373, 2011.

[3] Tobias Columbus. On the complexity of contraction hierarchies. *StudentâĂŹs thesis-Karlsruhe Institute of Technology-ITI Wagner*, 2009.

[4] Prajakta Desai, Seng W Loke, Aniruddha Desai, and Jack Singh. Multi-agent based vehicular congestion management. In *Intelligent Vehicles Symposium (IV), 2011 IEEE*, pages 1031–1036. IEEE, 2011.

[5] Edsger. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[6] Transport for London. Travel in london. http://content.tfl.gov.uk/travel-in-london-report-8.pdf, 2015.

[7] OpenStreetMap Foundation. *OpenStreetMap*. http://openstreetmap.org/.

[8] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Experimental Algorithms*, pages 319–333. Springer, 2008.

[9] A. V. Goldberg and C. Harrelson. Computing the shortest path: A* search meets graph theory, July 2004.

[10] Peter Karich. *GraphHopper Routing Engine*. http://graphhopper.com.

[11] Peter Karich. Introducing graphhopper, a fast and flexible route planner. https://graphhopper.com/public/slides/2014-locationtech.pdf, October 2014.

[12] Daniel Krajzewicz, Jakob Erdmann, Michael Behrisch, and Laura Bieker. Recent development and applications of SUMO - Simulation of Urban MObility. *International Journal On Advances in Systems and Measurements*, 5(3&4):128–138, December 2012.

[13] LordFokas, elonen, and ritchieGitHub. *NanoHttpd*. https://github.com/NanoHttpd/nanohttpd. (GitHub Users).

[14] Kai Nagel and Michael Schreckenberg. A cellular automaton model for freeway traffic. *Journal de Physique I*, 2, 1992.

[15] Valery Naumov and Thomas R Gross. Connectivity-aware routing (car) in vehicular ad-hoc networks. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*, pages 1919–1927. IEEE, 2007.

[16] Vi Tran Ngoc Nha, Soufiene Djahel, and John Murphy. A comparative study of vehicles' routing algorithms for route planning in smart cities. In *Vehicular Traffic Management for Smart Cities (VTM), 2012 First International Workshop on*, pages 1–6. IEEE, 2012.

[17] Aaron Ogle. Leaflet.animatedmarker on github. https://github.com/openplans/Leaflet.AnimatedMarker.

[18] Marcus Rickert, Kai Nagel, Michael Schreckenberg, and Andreas Latour. Two lane traffic simulations using cellular automata. *Physica A: Statistical Mechanics and its Applications*, 231(4):534–550, 1996.

[19] David Rohmer and Nathan Rajlich. *Java-WebSocket*. http://java-websocket.org/.

[20] Benno Tilch and Dirk Helbing. Evaluation of single vehicle data in dependence of the vehicle-type, lane, and site. In *Traffic and Granular FlowâĂŹ99*, pages 333–338. Springer, 2000.

[21] Peter Wagner, Kai Nagel, and Dietrich E Wolf. Realistic multi-lane traffic rules for cellular automata. *Physica A: Statistical Mechanics and its Applications*, 234(3):687–698, 1997.

[22] Shen Wang, Soufiene Djahel, and Jennifer McManis. A multi-agent based vehicles re-routing system for unexpected traffic congestion avoidance. In *Intelligent Transportation Systems (ITSC), 2014 IEEE 17th International Conference on*, pages 2541–2548. IEEE, 2014.

[23] Horst F Wedde, Sebastian Lehnhoff, Bernhard van Bonn, Zeynep Bay, Sven Becker, Sven Böttcher, Christian Brunner, Armin Büscher, Thomas Fürst, Anca M Lazarescu, et al. Highly dynamic and adaptive traffic congestion avoidance in real-time inspired by honey bee behavior. In *Mobilität und Echtzeit*, pages 21–31. Springer, 2007.

[24] Dietrich E Wolf. Cellular automata for traffic simulations. *Physica A: Statistical Mechanics and its Applications*, 263(1):438–451, 1999.

[25] Ning Wu. Verkehr auf schnellstraßen im fundamentaldiagramm. *Straßenverkehrstechnik*, 8:378–388, 2000.

[26] Haitao Zhang and Yanyan Li. Modeling and analysis of traffic guidance systems based on multi-agent. *International Journal of Control and Automation*, 7(2):21–32, 2014.