

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.

Thursday 28th April, 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, Thursday 28th April, 2016

Contents

1	Contextual Background	1
1.1	Routing in the 21st Century	1
1.2	Self-Driving Vehicles	1
1.3	Congestion	2
1.4	Simulation	2
2	Technical Background	5
2.1	Algorithmic Background	5
2.2	Implementation Background	8
3	Project Execution	13
3.1	Overview	13
3.2	Initial Implementation	13
3.3	Realistic Simulation	16
3.4	Performance and Architecture Improvements	19
3.5	Algorithm Design and Implementation	24
4	Critical Evaluation	25
5	Conclusion	27
A	An Example Appendix	31

List of Figures

2.1	Paths searched by each algorithm	6
2.2	Formation of traffic jams in the Nagel-Schreckenberg Model	8
2.3	An overview of the GraphHopper Architecture	10
2.4	OpenStreetMap way representation compared to edges in GraphHopper	10
3.1	The initial architecture for the simulation engine	14
3.2	Vehicle positions at different iterations	16
3.3	The problem with GraphHopper edges	17
3.4	The Nagel-Schreckenberg Model across edge boundaries.	18
3.5	Graphical glitches caused by displaying more than 16,000 images.	20
3.6	SVG Rendered orange markers showing 30,000 vehicles	20
3.7	The Vehicle and VehicleIterator class structure	22
3.8	Post-simulation analysis of the 9,000th iteration of 15,000 vehicles	24
3.9	VisualVM Profile of the most time consuming functions	24

List of Listings

2.1	GraphHopper request and response	11
3.1	The <code>slowStep</code> implementation making use of the <code>CellIterator</code>	19
3.2	The single-threaded <code>timestep</code> function	21

Executive Summary

A compulsory section, of at most 1 page

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) [5] is used for the raw mapping data. Specific sections of the maps can be downloaded from Geofabrik.
- The Open Source routing engine GraphHopper [6] was used for performing basic routing requests and handling storage and processing of the OpenStreetMaps data.
- NanoHttpd [7] was used for a static file server to provide HTML, CSS and images to the front end.
- Java WebSocket [10] 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 [3] 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 location to location routing requests
VANET	:	Vehicular-AdHoc NETwork
V2V	:	Vehicle to Vehicle
V2I	:	Vehicle to Vehicle

Acknowledgements

An optional section, of at most 1 page

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 as well as knowledge gained through experience. MapQuest was the first online service to change that - instead of figuring out a route yourself, you could enter your location and destination and have a route provided to you. Unfortunately, using these routes meant printing out or writing down the instructions yourself - the route was fixed from when it was calculated.

In 2005, the first online version of Google Maps was released. Unlike its predecessors, Google Maps used real-time traffic analysis to improve the routes it offered to its users. However, with the exception of high-end vehicles with GPS navigation, these were still usually printed for use.

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 whilst travelling, whilst Google could use this information to improve their data on congestion.

1.2 Self-Driving Vehicles

This technology has now been integrated into almost every modern vehicle available to purchase today. GPS based navigation, with traffic information and turn by turn routing instructions are standard features in many car models. Furthermore, we are seeing some further intelligence being added to cars, from two perspectives.

The first is modifying existing cars - adding features such as cruise control, automatic braking systems, reading of road signs, lane following on highways, and so on until eventually 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 enables further automation by the car itself. The second approach is from companies like Google, who have been building a self-driving vehicle ‘from scratch’, creating a vehicle that has no steering wheel and requires no human intervention to drive from one location to another.

1.3 Congestion

With the advent of Google Maps and the proliferation of mobile devices, many drivers are now provided with directions that can incorporate large amounts of additional information - including current traffic conditions and road closures. This is particularly important in cities, where the levels of congestion during ‘rush hour’ can have a drastic impact on travel time.

Looking forwards, we can see two trends that suggest that congestion will become more of an issue in future. Firstly, more and more people are living in cities - even with good public transportation, this will increase the number of vehicles on the road even if it lowers the proportion of households that own cars. Secondly, on-demand transport solutions (such as those provided by Uber) are leading to more vehicles on the road, especially at peak times. One of Uber’s key insights is using market techniques to better match supply and demand 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 paying riders. The net result of this is that high-demand in certain areas will create further congestion, even though on demand transportation is usually more efficient than personal car ownership.

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 raises many questions about the role of personal transportation and the effect this will have on congestion. Will driving become as dated as horseriding is today - or will people’s enjoyment of it mean that there’ll always be human driven vehicles on the road? More importantly, will self driving cars improve or harm congestion - and how will they 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

One way of finding answers to these questions is simulation - modelling and making assumptions about future behaviours and analysing the results. Vehicles 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 (e.g busses, 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 Types of Simulation

V2V vs V2I etc

There are two main tools in use today for running vehicle simulations. The first is MATSim [1], and the second is SUMO [2].

1.4.2 MATSim

1.4.3 SUMO

Existing tools are cumbersome, slow, non-interactive and designed to be used for any type of multi-agent problem - from routing air traffic to planning for evacuation in crisis situations. This project aims to create a system designed exclusively for one goal - road based simulations for vehicles and traffic. By focussing on a specific use case, a tool can be easier to use, faster to setup and provide specialised functionalities for our use case.

Additionally, existing tools tend to focus on simulation then visualisation, without combining the two. Simultaneous simulation and visualisation allows a much tighter loop of testing and iterating on algorithm design and analysis.

The high-level objective of this project is to build an easier, faster way of simulating the behaviour of vehicles, primarily in cities. We will use this simulation engine to design and improve a potential algorithm for multi-vehicle routing. The concrete aims 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. Demonstrate the power and flexibility of the engine by experimentally creating a novel multi-vehicle routing algorithm.

Chapter 2

Technical Background

2.1 Algorithmic Background

2.1.1 Dijkstra's Algorithm

Dijkstra's Algorithm [4], originally designed in 1956, forms the foundation of most modern routing algorithms.

```
1 Function DIJKSTRA( $G, s, d$ )
2    $Q \leftarrow$  new Priority Queue
3    $dist[s] \leftarrow 0$ 
4   for vertex  $v \neq s \in G$  do
5      $dist[v] = \infty$ 
6      $prev[v] = undefined$ 
7      $Q.add(v, dist[v])$  // Adds  $v$  to priority queue with weight  $dist[v]$ 
8   end
9   while  $Q \neq \emptyset$  do
10     $u \leftarrow Q.extract\_min()$ 
11    if  $u = d$  then
12      | Return BuildPath( $u$ )
13    end
14    for neighbour  $v$  of  $u$  do
15       $d \leftarrow dist[u] + \text{distance}(u, v)$ 
16      if  $d < dist[v]$  then
17        |  $dist[v] \leftarrow d$ 
18        |  $prev[v] \leftarrow u$ 
19        |  $Q.decrease\_priority(v, d)$ 
20      end
21    end
22  end
```

Algorithm 2.1: Dijkstra's Algorithm

The algorithm operates on a graph consisting of nodes connected to each other by edges. Each edge has a weight - this could be the length of the road in a real world map. Given two nodes S and D , the goal is to return a list of edges that represents the shortest path between the nodes.

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

Algorithm 2.1 shows how it works in pseudocode. The implementation for the **BuildPath** function simply iterates through each nodes parent and adds it to the output list. We also define the **distance** function that simply returns the weight of the edge between vertices u and v .

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.

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 in the opposite direction from the destination.

In Dijkstra's algorithm, the distance from the source node is used as the key in the priority queue (line X). The A* algorithm instead seeks to minimize the function $f(v) = dist[v] + h(v)$, where $h(v)$ is a heuristic that estimates the cost to destination.

A bidirectional version of the A* algorithm can also be used for further improvements to performance.

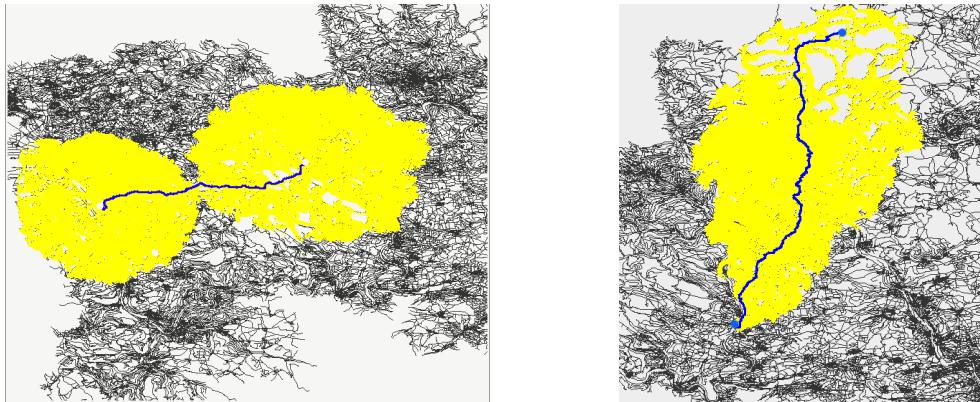


Figure 2.1: Paths searched by each algorithm

Contraction Heirarchies

If the weights on each edge of the graph are known in advance, we can process the graph to pre-compute the shortest path between certain key nodes in the graph. One such algorithm for this is the Contraction Heirarchies algorithm, which works by building shortcuts on top of each other in a heirarchy. It then uses a modified version of 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 Heirarchies 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. **Cite:** <https://graphhopper.com/public/slides/2014-locationtech.pdf>

2.1.2 Multi-Vehicle Routing Algorithms

Dealing with the routing for a single vehicle is now essentially a solved problem, with various implementations used in many commercial products for consumers and businesses. However, the problem of routing multiple vehicles simultaneously is a problem with many solutions, but no accepted ‘best practice’ technique.

Multi-Vehicle routing algorithms come in various forms, depending on the author’s expectations for future vehicle connectivity. Some algorithms are designed to have a large scale overview of where each vehicle is on the road, 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.

The two types of algorithm are generally referred to as **V2V** (Vehicle to Vehicle) or **V2I** (Vehicle to Infrastructure).

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 uses of a VANET. VANETs have a number of potential use cases - allowing vehicles to follow one another with no driver intervention, realtime calculation and distribution of traffic information and so on.

Algorithms for multi-vehicle routing making use of VANETs have also been created.

Modern simulation engines are often designed to support both V2V and V2I algorithms - these engines are known as V2X engines.

2.1.3 Nagel-Schreckenberg Model

The Nagel-Schreckenberg Model [8] 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, in parallel to simulate the flow of traffic. Each car has a fixed velocity v for that step, which represents the number of cells the vehicle will move forward in the final rule. The model does not allow for overtaking, and as a result exhibits 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. **Randomization** - reduce the velocity by 1 with probability p .
4. **Car 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.

We’ll now run through a brief example of the effects of each step. Our road will have 9 cells with two vehicles on them. Our max speed (v_{max}) will be 5. For this example, we will ignore the randomization step.

	4				0		
--	---	--	--	--	---	--	--

We now perform the acceleration step for each vehicle. The second cell vehicle is currently at speed 4 - as the vehicle in front is only 3 cells away, it does not accelerate. The sixth cell vehicle increments its speed from 0 to 1.

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

				3		1	
--	--	--	--	---	--	---	--

Finally, the vehicles move to their new destinations. This demonstration briefly shows the behaviour of two vehicles, but fails to demonstrate the creation and flow of traffic jams. In Figure 2.2, 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.

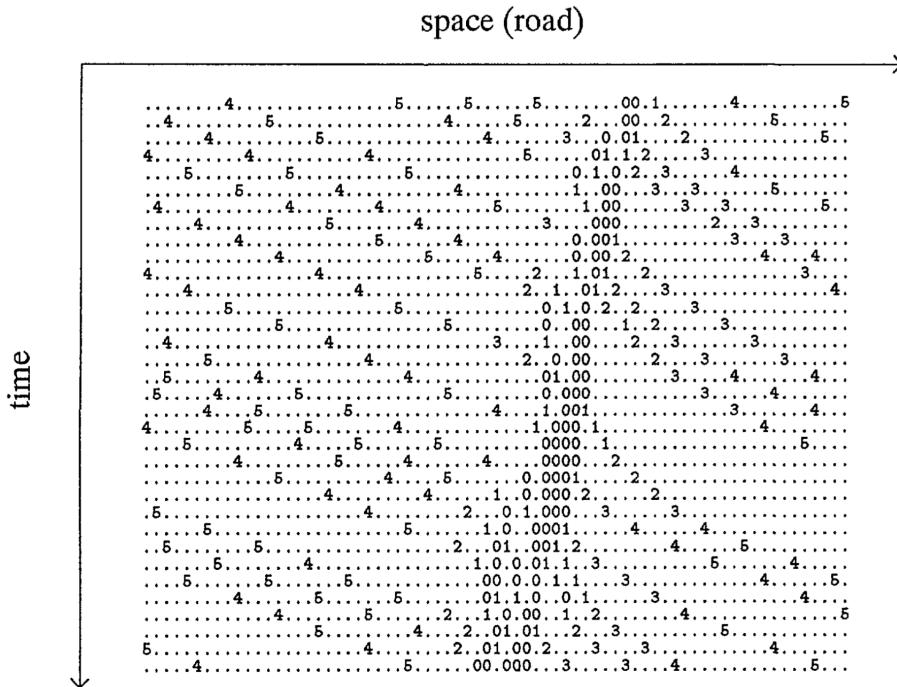


Figure 2.2: Formation of traffic jams in the Nagel-Schreckenberg Model

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 flexible 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 2.3 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 2.4 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

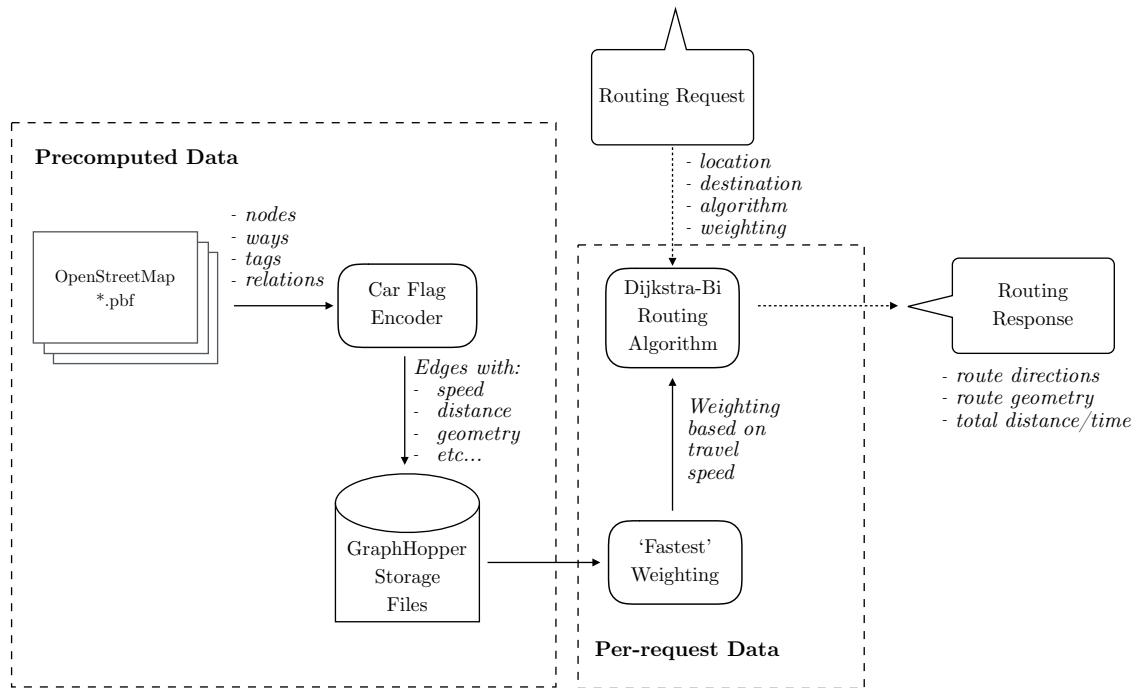


Figure 2.3: An overview of the GraphHopper Architecture



Figure 2.4: 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 `PathWrappers` 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 and 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 flexible enough to deal with the challenges of algorithm design.
4. **Algorithm Design and Implementation** - creating and improving 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 the primary objective of having 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 to the engine, it 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 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.

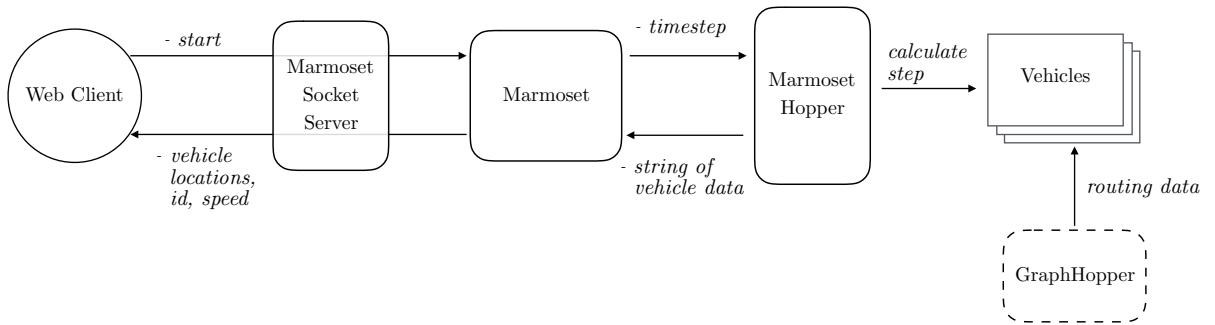


Figure 3.1: The initial architecture for the simulation engine

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, I created a GitHub fork of the project and added 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, I created my own shell script for building and running the marmoset engine. It supports four main actions - clean, build, rebuild and run. It also allows multiple commands to be run in succession for convenience.

Additionally, I wanted to create a modern codebase in spite of the core engine being written in Java. Although I briefly explored using Scala, 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.

<pre> 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(); } </pre> <p style="text-align: center;">Java 7 Implementation</p>	<pre> public String getVehicleString() { return vehicles.stream() .map(Vehicle::toString) .collect(Collectors.joining(",")); } </pre> <p style="text-align: center;">Java 8 Implementation</p>
--	--

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, I used the NanoHtt [7] library, as it requires minimal setup and is easy to run on a separate thread. For the client-server communication, I initially used the NanoHttpd WebSocket implementation, but it did not appear to be fully functional. As such, I switched to using the Java-Websocket library [10], extending the WebSocketServer class to create the MarmosetSocketServer class.

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

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 that 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, I had not established how well the front or back end would perform. As a way of simplifying the data flow and processing, `Marmoset` simply waits one second between each call of the timestep function. As both front end and back end take significantly less time 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 very inaccurate, 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 centered on the correct location. 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 [3] 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 [9] library handles smoothly moving the points to their next location, making the cars look like they're 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 Cars 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 a high volume of vehicles driving on them.

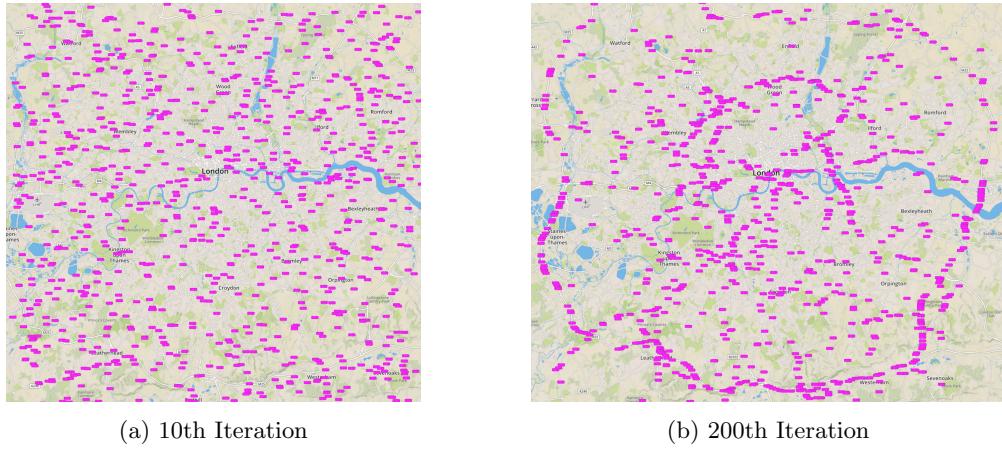


Figure 3.2: Vehicle positions at different iterations

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 curved path would take much longer to travel than a straight one.
- Vehicles are independent. Vehicles travel 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.

For the next stage of development, I focussed 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 dealing with the integration with 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.

3.3. REALISTIC SIMULATION

Ultimately, I chose to use GraphHopper's edges as the base unit for cells. This is 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 is 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, there 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 values. This is likely to harm cache performance, something particularly important given the frequency that the cells will be accessed.

Instead, we can simply allocate a two-dimensional array of booleans 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 or reversed. When a route is provided by the routing engine, it is in the form of a list of `EdgeIteratorState`. 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 in. The end result of this is that every edge appears to be a forward edge when received from the engine. 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.

In Figure 3.3, we can see why this may be an issue. When moving through the edges, we are unable to tell if we are travelling forwards or backwards based on the `EdgeIteratorState` by itself - the ID is the same and the `isForward` method returns true in both directions. Initially, I had thought 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 the cell array. However, it is important to note that we don't actually 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 above.

The technique I came up with 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.

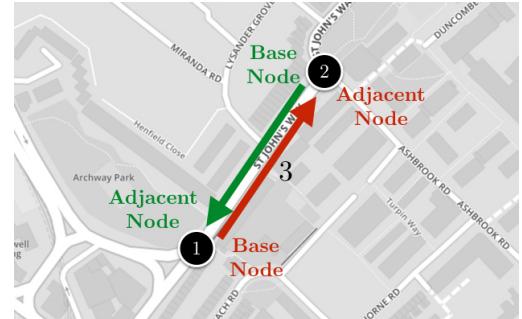


Figure 3.3: The problem with GraphHopper edges

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.

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`.

3.3.2 Vehicle and Cell Iterators

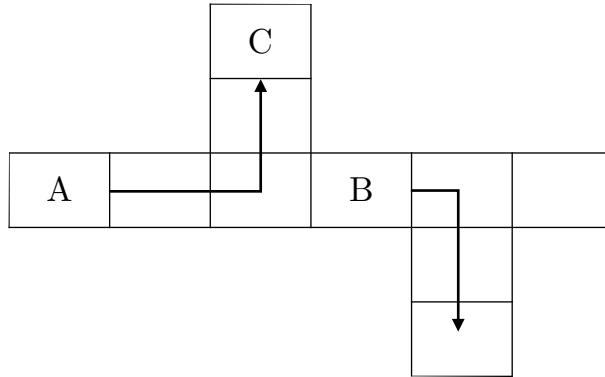


Figure 3.4: The Nagel-Schreckenberg Model across edge boundaries.

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. However, 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.

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 vehicle class 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 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

`;CREATE ARCHITECTURE DIAGRAM YAH;`

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.

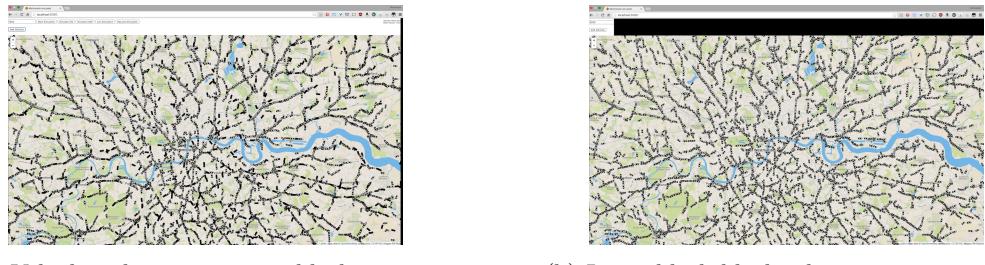
- The back end still waits one second per iteration.
- The back end architecture does not allow for vehicles with different behaviour.
- The back end does not make optimal use of the multiple threads and cores that modern computers have available to them.
- The front end has graphical glitches and poor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

I had initially used the Leaflet.AnimatedMarker library to make it appear as though the vehicles are 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 I used was reasonably large and in colour. I created a much smaller vehicle image (from 15KB to 1KB.), which did improve 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.

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. I 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 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.



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

Instead, I decided to experiment with SVG (Scalable Vector Graphics), an XML based web API. 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 Animated and normal image markers 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 backend initially just sent down data in string format - for example, `10067—51.5306611887—0.1609468558—1` for vehicle with id 10067. This format has a number of issues. 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 timelock

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

```
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. However, 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. Minor changes were made to ensure that the filter step for 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 is thread-safe for making routing requests is an easy way to further improve performance.

EXPLAIN THIS BIT FURTHER MAYBE???

3.4.3 Architecture Improvements

One of the issues with the existing system is the lack of support for multiple vehicle types - although there is a Vehicle class that could be subclassed, it is not easy to modify its behaviour without having to rewrite much of the code. As such, I decided a refactor of this part of the system was required to improve the architecture of the system.

Vehicle Refactoring

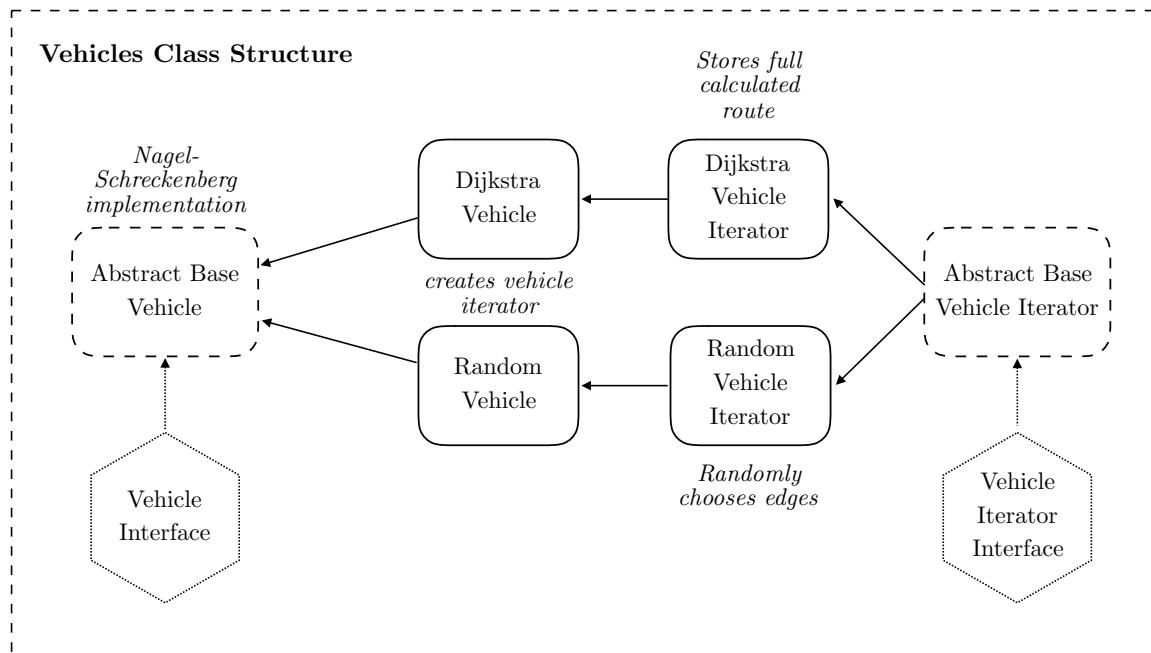


Figure 3.7: The Vehicle and VehicleIterator class structure

Following the refactor, the Vehicle and VehicleIterator have been changed into interfaces with the public methods originally implemented in the Vehicles 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, I also created a `RandomVehicle` and corresponding `RandomVehicleIterator` that simply drive around at random with no overarching behaviour.

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 any point in time.

With the goal of understanding the levels of congestion, I tracked two main vehicle metrics: the number 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, whilst a static method returns the header for the CSV file to output.

Route Metrics

We can also better understand congestion by looking at how long a journey should take compared to how long it did take. The `GraphHopper` `GHResponse` class stores the length of time a route is expected to take. 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, I added a `printMetrics` function to `BaseVehicle` and provided implementations in `DijkstraVehicle`, which 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 that holds 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.

The code previously used to convert the `Vehicle` objects into strings was repurposed and instead placed into an iteration file, meaning the data is human readable and easy to parse. Adding a simple input box

with basic parsing code allows the data to be explored interactively after the simulation has finished, as can be seen in Figure 3.8.

3.4.5 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.

A command line flag (`--file`) was added representing the number of vehicles to be simulated. The simulation creates a folder for the metrics including the number of vehicles and a Unix timestamp, allowing it to be uniquely identified.

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

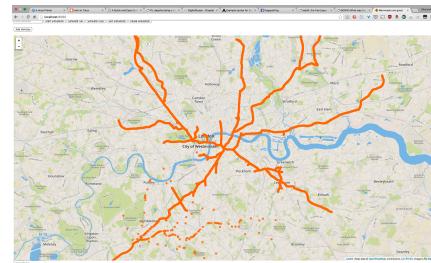


Figure 3.8: Post-simulation analysis of the 9,000th iteration of 15,000 vehicles

Hot Spots – Method	Self Time [%]	Self Time	Self Time (CPU)	Total Time	Total Time (CPU)
com.graphhopper.marmoset.vehicle.BaseVehicle.updateLocation ()	24.7%	2,196 ms	2,196 ms	2,392 ms	2,392 ms
com.graphhopper.marmoset.vehicle.BaseVehicle.addToBuffer ()	6.8%	603 ms	603 ms	703 ms	703 ms
com.graphhopper.marmoset.vehicle.DijkstraVehicleIterator.duplicate ()	3.2%	279 ms	279 ms	279 ms	279 ms
com.graphhopper.marmoset.MarmosetHopper\$\$Lambda\$15.1276042862.accept ()	1.2%	102 ms	102 ms	2,494 ms	2,494 ms
com.graphhopper.marmoset.MarmosetHopper.timestep ()	1.1%	93.3 ms	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, meaning simulations with large numbers of vehicles could be simulated to completion in a reasonably short amount of time.

3.5 Algorithm Design and Implementation

Test

Chapter 4

Critical Evaluation

Chapter 5

Conclusion

Bibliography

- [1] MatSim vehicle simulation engine - incomplete.
- [2] Sumo vehicle simulation engine - incomplete.
- [3] Vladimir Agafonkin, Dave Leaver, et al. *Leaflet.js*. <http://leafletjs.com/>.
- [4] Edsger. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
- [5] OpenStreetMap Foundation. *OpenStreetMap*. <http://openstreetmap.org/>.
- [6] Peter Karich. *GraphHopper Routing Engine*. <http://graphhopper.com>.
- [7] LordFokas, elonen, and ritchieGitHub. *NanoHttpd*. <https://github.com/NanoHttpd/nanohttpd>. (GitHub Users).
- [8] Kai Nagel and Michael Schreckenberg. A cellular automaton model for freeway traffic. *Journal de Physique I*, 2, 1992.
- [9] Aaron Ogle. Leaflet.animatedmarker on github. <https://github.com/openplans/Leaflet.AnimatedMarker>.
- [10] David Rohmer and Nathan Rajlich. *Java-WebSocket*. <http://java-websocket.org/>.

Appendix A

An Example Appendix