Dissertation Type: research

DEPARTMENT OF COMPUTER SCIENCE

MARMOSET: Multi Agent Real-time Multi-core Online Simulation for Effic

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.

Tuesday 19th 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, Tuesday 19th April, 2016

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

Executive Summary

A compulsory section, of at most 1 **page**

This section should précis the project context, aims and objectives, and main contributions (e.g., deliverables) and achievements; the same section may be called an abstract elsewhere. The goal is to ensure the reader is clear about what the topic is, what you have done within this topic, *and* what your view of the outcome is.

The former aspects should be guided by your specification: essentially this section is a (very) short version of what is typically the first chapter. Note that for research-type projects, this **must** include a clear research hypothesis. This will obviously differ significantly for each project, but an example might be as follows:

> My research hypothesis is that a suitable genetic algorithm will yield more accurate results (when applied to the standard ACME data set) than the algorithm proposed by Jones and Smith, while also executing in less time.

The latter aspects should (ideally) be presented as a concise, factual bullet point list. Again the points will differ for each project, but an might be as follows:

- I spent 120 hours collecting material on and learning about the Java garbage-collection sub-system.
- I wrote a total of 5000 lines of source code, comprising a Linux device driver for a robot (in C) and a GUI (in Java) that is used to control it.
- I designed a new algorithm for computing the non-linear mapping from A-space to B-space using a genetic algorithm, see page 17.
- I implemented a version of the algorithm proposed by Jones and Smith in [6], see page 12, corrected a mistake in it, and compared the results with several alternatives.

Supporting Technologies

A compulsory section, of at most 1 page

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) [3] is used for the raw mapping data. Specific sections of the maps can be downloaded from Geofabrik.
- The Open Source routing engine GraphHopper [4] was used for performing basic routing requests and handling storage and processing of the OpenStreetMaps data.
- NanoHttpd [5] was used for a static file server to provide HTML, CSS and images to the front end.
- Java WebSocket [7] 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

An optional section, of roughly 1 or 2 pages

Any well written document will introduce notation and acronyms before their use, *even if* they are standard in some way: this ensures any reader can understand the resulting self-contained content.

Said introduction can exist within the dissertation itself, wherever that is appropriate. For an acronym, this is typically achieved at the first point of use via "Advanced Encryption Standard (AES)" or similar, noting the capitalisation of relevant letters. However, it can be useful to include an additional, dedicated list at the start of the dissertation; the advantage of doing so is that you cannot mistakenly use an acronym before defining it. A limited example is as follows:

OSM : OpenStreetMap

Acknowledgements

An optional section, of at most 1 **page**

It is common practice (although totally optional) to acknowledge any third-party advice, contribution or influence you have found useful during your work. Examples include support from friends or family, the input of your Supervisor and/or Advisor, external organisations or persons who have supplied resources of some kind (e.g., funding, advice or time), and so on.

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 šurge 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.

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 simulations.

The high-level objective of this project is to build an easier and faster way of simulating the behaviour of vehicles, primarily in cities. 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 [2], originally designed in 1956, forms the foundation of most modern routing algorithms.

```
Function DIJKSTRA(G,s,d)
    Q ← new Priority Queue
    dist[S] ← 0
    for vertex v ∈ G do
        dist[v] = ∞
        prev[v] = undefined
        Q.add(v, dist[v])
    end
    while Q ≠ ∅ do
        u ← Q.extract_min()
        if u = d then
            Return BuildPath(u)
        end
        for neighbour v of u do
            d ← dist[u] + distance(u, v)
            if d < dist[v] then
                dist[v] ← d
                prev[v] ← u
                Q.decrease_priority(v, d)
            end
        end
    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.

A* Algorithm

By itself, Dijkstra's algorithm is

Contraction Heirarchies

2.1.2 Nagel-Schreckenberg Model

The Nagel-Schreckenberg Model [6] 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 breifly shows the behaviour of two vehicles, but fails to demonstrate the creation and flow of traffic jams. In Figure 2.1, 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.

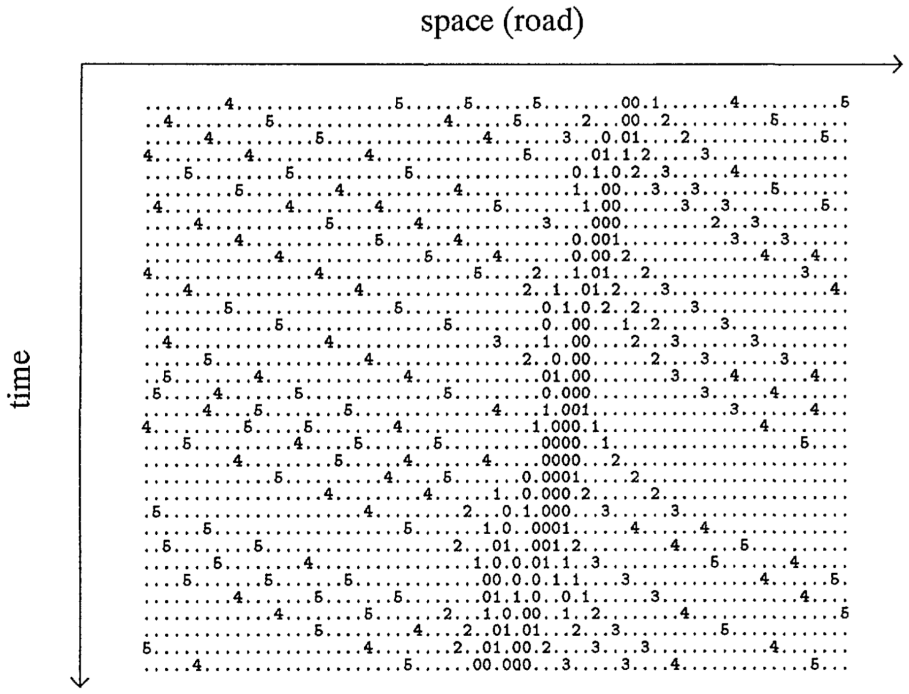

Figure 2.1: Formation of traffic jams in the Nagel-Schreckenberg Model

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 (primarly 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 graph for performance improvements, supporting multiple dijkstra and A* algorithms and running a web server for making requests. Additionally, it has built in support for car, truck, motorcycle, bicycle and other types of travel as well as the ability to create custom vehicle types.

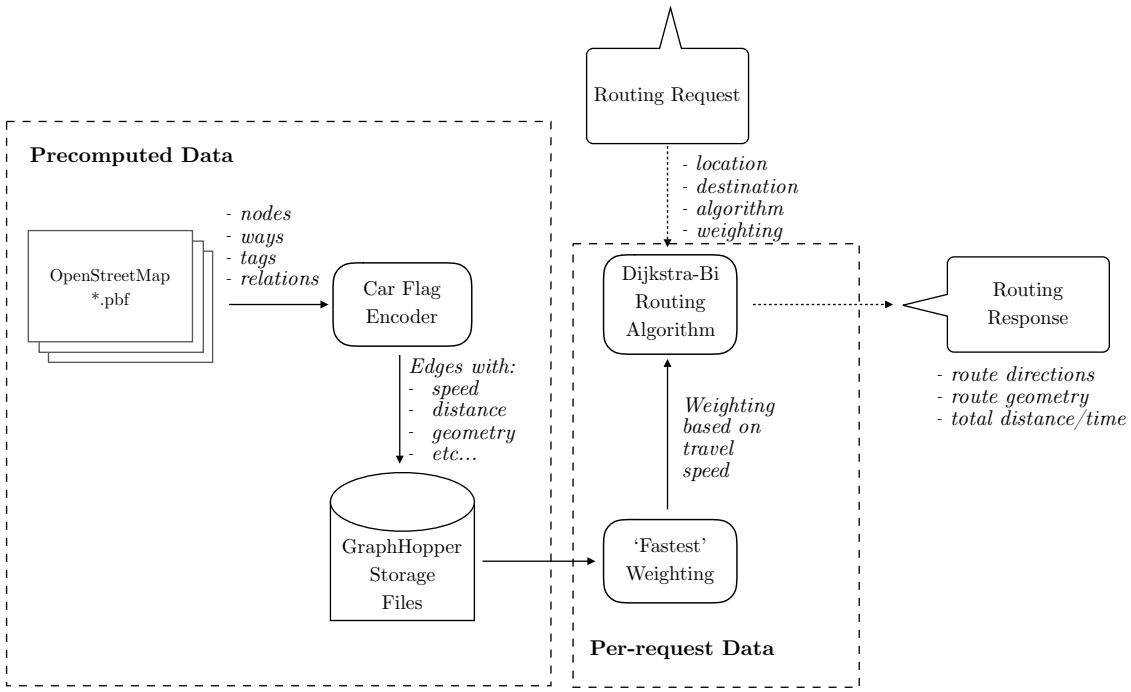

Figure 2.2: An overview of the GraphHopper Architecture

Figure 2.2 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.

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 2.3 shows the difference between ways, roads and edges. We can see that Holloway Road extends beyond the OSM way in 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 multiple edges whenever there is a junction of any kind - these can be seen in orange. We can now perform the routing calculation mentioned before - there is now an edge along a subsection of Holloway Road that conncets Hargrave Road to St John's Villas.

Figure 2.3: OpenStreetMap way representation compared to edges in GraphHopper

Flag Encoders

Internally, GrahHopper uses a clever compression trick to efficiently store the road data. 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 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 and static. 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.

Chapter 3

Project Execution

3.1 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 a new module for the project, called marmoset.

This allowed me to use GraphHopper as a library for the majority of tasks, but was still able to modify the visibility of certain methods that were necessary for the tasks I wanted to perform.

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 separation 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.

```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 Initial Client-Server setup

There are two key parts of the simulation engine - a front end visualisation and a back end simulation. The first thing to do was create a server and find an efficient way of sending simulation data to the front end.

For the raw file server, I used the NanoHttd [5] library, as it required 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 [7], extending the WebSocketServer class to implement my own protocol on top of WebSocket.

3.2.1 Initial API design

3.2.2 Routing Behaviour

3.2.3 Multi-Vehicle Support

3.3 Cell Automata Model

Test

3.3.1 Vehicle and Cell Iterators

Test

3.4 Performance Improvements

Test

3.4.1 Binary vs String data

Test

3.4.2 Removing animations

Test

3.4.3 Removing timelock

Test

3.4.4 DOM to Canvas

Test

3.4.5 Parallel processing

Test

3.4.6 Metric calculations

Chapter 4

Critical Evaluation

A topic-specific chapter, of roughly 15 **pages**

This chapter is intended to evaluate what you did. The content is highly topic-specific, but for many projects will have flavours of the following:

1. functional testing, including analysis and explanation of failure cases,
2. behavioural testing, often including analysis of any results that draw some form of conclusion wrt. the aims and objectives, and
3. evaluation of options and decisions within the project, and/or a comparison with alternatives.

This chapter often acts to differentiate project quality: even if the work completed is of a high technical quality, critical yet objective evaluation and comparison of the outcomes is crucial. In essence, the reader wants to learn something, so the worst examples amount to simple statements of fact (e.g., "graph X shows the result is Y"); the best examples are analytical and exploratory (e.g., "graph X shows the result is Y, which means Z; this contradicts [1], which may be because I use a different assumption"). As such, both positive *and* negative outcomes are valid *if* presented in a suitable manner.

Chapter 5

Conclusion

A compulsory chapter, of roughly 5 pages

The concluding chapter of a dissertation is often underutilised because it is too often left too close to the deadline: it is important to allocation enough attention. Ideally, the chapter will consist of three parts:

1. (Re)summarise the main contributions and achievements, in essence summing up the content.
2. Clearly state the current project status (e.g., "X is working, Y is not") and evaluate what has been achieved with respect to the initial aims and objectives (e.g., "I completed aim X outlined previously, the evidence for this is within Chapter Y"). There is no problem including aims which were not completed, but it is important to evaluate and/or justify why this is the case.
3. Outline any open problems or future plans. Rather than treat this only as an exercise in what you *could* have done given more time, try to focus on any unexplored options or interesting outcomes (e.g., "my experiment for X gave counter-intuitive results, this could be because Y and would form an interesting area for further study" or "users found feature Z of my software difficult to use, which is obvious in hindsight but not during at design stage; to resolve this, I could clearly apply the technique of Smith [7]").

Bibliography

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

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

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

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

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

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

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

Appendix A

An Example Appendix

Content which is not central to, but may enhance the dissertation can be included in one or more appendices; examples include, but are not limited to

- lengthy mathematical proofs, numerical or graphical results which are summarised in the main body,
- sample or example calculations, and
- results of user studies or questionnaires.

Note that in line with most research conferences, the marking panel is not obliged to read such appendices.