I, Henry Chadban, solemnly and sincerely swear that this thesis was entirely my own work except where otherwise acknowledged. A summary of my contributions and where my work depend on others is included below.

- I encountered and developed the initial conceptual ideas underlying this thesis myself.

- I undertook a literature review myself in an effort to better understand previous approaches to the field of public transport optimisation. This can be found in Chapter 2 of this thesis.

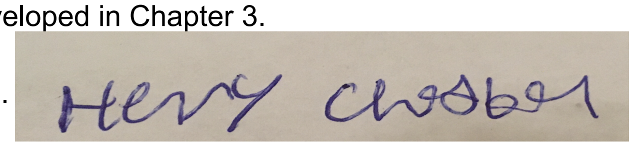
- On the suggestion of my supervisor Dr Guodong Shi, I decided to switch from focusing on developing public transport optimisers to developing a system which could evaluate public transport optimisers developed by others.

- I developed the formal overview of both the problem to be solved and my high-level approach to solving it. This can be found in chapter 1 and 3 respectively.

- I implemented my solution in software. I wrote the software myself in the Python program language. My software made use of numerous open-source libraries, all of which are cited in the bibliography as well as in the code itself.

- I evaluated my model using information about the Sydney trains network. Passenger journey data is made available by Transport for NSW, station-station trip times and a basic timetable were determined using the trip-view app. Geographical locations of the stations were extracted from Google Maps. All this information was collected by myself using these publicly available sources.

- The basic optimiser used in the evaluation of my system was developed my myself based on my own original concept which is developed in Chapter 3.

The above is an accurate statement of my contribution. 

Development of Computational Methods for Public Transport Schedule Optimisation and Evaluation

HENRY CHADBAN

B.Eng (Hons)

THE UNIVERSITY OF SYDNEY

Supervisor: Dr Guodong Shi

A thesis progress report submitted in fulfilment of
the requirements for the degree of
Bachelor of Engineering (Honours)

School of Aerospace, Mechanical and Mechatronic Engineering
Faculty of Engineering
The University of Sydney
Australia

8 November 2022

Abstract

Efficient methods of scheduling public-transport services can simultaneously reduce wait and travel times for passengers and financial costs for the system operators. While the general problem is NP hard and hence computationally intractable, simplifications and approximations have allowed the development of schedule optimisers which are able to generate high quality timetables for public transport.

However most of the existing literature only applies these optimisers to small scale "toy" transport networks, which are not comparable to the large, complex public transport networks in a major city. In this thesis we developed a software framework which can efficiently simulate a large scale public transport network, and automatically provide an evaluation of any particular schedule or schedule optimiser on that network in terms of operating cost and passenger travels. This will allow for the quality of particular public transport schedules or schedule optimisers to be easily evaluated on large-scale complex networks which more accurately reflect a real public transport system.

We evaluated our framework on a model of the Sydney Trains Network, using publicly available geographical and passenger volume information. This was done using both a simplified version of the real Sydney trains timetable, as well as with a timetable generated through a very simple optimisation process. We expect it would be straightforward to adapt our framework to other public transport systems and more complex optimisation approaches.

Acknowledgements

Thanks to my thesis supervisor Dr Guodong Shi for his advice about what topic areas would be most useful to focus on, as well as advice about the thesis process. I also wish to acknowledge Dr Benjy Marks, who developed the Latex template used for this document [Marks 2020].

Issac Newton once said "If I have seen further, it is by standing on the shoulders of Giants". While my work in this thesis is not of the same stature as Newtons, the same applies here. My work would not have been possible without the contributions of numerous others in the academic and open source software community. I am particularly grateful to the developers of the Python language and it's many libraries, without which the development of my framework would have been much more difficult. Full details of these libraries are of course included in the bibliography and where relevant in the text.

On a more personal note, I also wish to acknowledge my family, particularly my parents Martin and Kylie and my elder sister Lily. While they had no direct role in the creation of this work, without their continued moral support it would nonetheless have been impossible.

Disclosure

This thesis focuses on developing a software framework for evaluating public optimisers, with an evaluation performed on the Sydney Trains system.

While the author is a resident of Sydney and a frequent user of the Sydney Trains, I am not an employee of or in anyway affiliated with Sydney Trains or Transport for New South Wales (NSW). I am also not employed by or affiliated with any other transport related organisations, either in New South Wales, Australia or the rest of the World. While I am employed in a casual capacity by the NSW Departments of Health and Education, my work is unrelated to anything discussed in this thesis. This work was done in my capacity as a private citizen and as a student of the University of Sydney.

Any views expressed herein are my own, and I retain all rights legal and moral over this work. Furthermore it should be noted that the analysis of the Sydney Trains system presented, while extensive, still included numerous simplifications which make it insufficient to directly evaluate the performance of the real world system. Instead the analysis presented serves purely as a demonstration of how many framework could be used to evaluate a complex public transport system.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction and Problem Definition

1.1 Introduction to the Public Transport Problem

Having a transportation system able to transport people from place to place quickly and safely is of vital importance to modern society. Public transport systems such as train and bus networks are essential to the functioning of major cities, as they allow for rapid transportation without the extremely high financial, environmental and congestion costs of transport by private automobile. Public transport is also essential for those who cannot afford private automobiles, or are too young/old/disabled to drive.

When developing a public transport schedule, there is a fundamental tradeoff between the benefits (utility) to passengers and the financial cost to the system operator. For example, more frequent services reduce passenger waiting times and crowding and hence increase the benefit (utility) of the service to passengers. However more frequent services also increase the financial cost of providing the service as additional vehicles will need to be purchased and maintained, additional divers will need to be hired and additional energy will be consumed.

A good public transport schedule should obtain a good trade-off between the financial cost of system operation and the utility to passengers.

1.2 Problem Definition

Many optimisers exist which use various approaches to try and balance these two competing needs. However most of the literature relating to optimisers only applies them on small scale transport networks, which are not comparable to the large, complex public transport networks in a major city. For these optimisers to be truly useful, they must be able to be easily integrated with a simulator of the large complex transport network being analysed. Furthermore both the optimiser and simulator should be built in a way that allows them to be easily applied to different transport networks, and in a way that allows the end results to be easily interpreted.

This thesis aims to build a software framework which integrates these three aspects of public transport planning. Scheduling, Simulation and Evaluation. This will include developing a common framework to represent the physical network constraints and passenger travel patterns of a transport network. It will include developing a common framework for a public transport schedule, and implementing existing optimisation algorithms in a standardised way to produce the desired schedule from the provided network constraints and passenger travel patterns. It will also involve building a simulation of a public transport system which will be able to handle systems of comparable complexity to real world public transport systems. Equally essential is a framework for evaluating the results of the simulation in a way that is accessible and understandable to the end user.

For all components of the developed framework, it is essential that it is easy to understand and utilise. This will allow public transport planners, academics and interested laypeople to utilise this framework to evaluate their own transport networks or optimisers. An end goal of this thesis will be to use the framework to implement several different optimisation techniques, and evaluate their performance on a model of a large complex urban public transport system. This was done using the Sydney Trains network, as it is the one which the author is most relevant and familiar.

1.3 Introduction to Nomenclature

In this project we will be using a common nomenclature to represent elements common to all types of public transport systems. This is because we wish to develop a software package which is useful in optimising all types of public transport systems, whether they are rail networks, bus networks, ferry networks, passenger air networks or potentially even mail, rail or sea freight networks. Our methods and eventual software package should be equally applicable to any sort of transport system where objects to be moved (whether they be passengers, mail, containers) are "pooled" at origin and destinations and travel between these destinations in common "public" vehicles. Hence will we use nomenclature which is neutral between different types of transport systems.

An overview of nomenclature used in this thesis is listed below.

- **Agents** The objects which which are being moved around by the transport system. These would normally be actual passengers but could potentially be cargo. It should be noted that for computational efficiency reasons it is more efficient to clump multiple passengers going between the same origin and destination pair. Hence in the software one agent will often represent multiple different passengers.
- **Nodes** Origins and destinations between which agents wish to travel. Eg train stations, bus stops.
- **Origin** The node at which an agent starts its journeys.
- **Destination** The node which an agent wishes to reach during it's journey.
- **Vehicles** The physical vehicle in which passengers travel from node to node. Eg trains, buses.

- **Edges** The physical connection between two nodes, which takes some finite non-zero amount of time for a vehicle to travel along. Eg a train line or a road along which a bus could travel.
- **Headway** How long must the "gap" be between two vehicles travelling along a particular edge. Eg if the headway along an edge is two minutes, then after one vehicle starts travelling along an edge another vehicle must wait two minutes before it can start travelling along that edge. The headway of an edge is determined by how closely together vehicles can safely operate.
- **Route** The series of edges and nodes along which a particular vehicle travels as part of a contiguous journey. For example a train line or a bus route. This does not necessary correspond to physical infrastructure. For example a single train track with both an "all-stations" service and a "express" service that only stops at a few stations would be considered to be two separate routes under this approach. Additionally a bidirectional train line where one can catch trains going from both A to B or B to A would also considered to be two separate routes.
- **Service** The process of an individual vehicle moving along a particular route at a particular time. Eg A vehicle travelling along route A at 9.45am would be a Service.
- **Journey** The combination of services which an agent must use to travel from it's Origin and Destination.
- **Journey Time** The time it takes for an agent to achieve it's journey. All journey times in this thesis are provided in minutes.
- **Pathfinding** The process by which the fastest journey an agent can make between their origin and destination is determined.
- **Utility** A general term for non-financial benefit commonly used in Economics. In the context of our thesis it used to refer to the benefit passengers obtain from shorter journey times.

- **Disutility** The opposite of utility. This is also referred to as the "Passenger Time Cost". It is calculated by multiplying the time taken by passengers to make their journey by the value of the passengers time
- **Marginal Costs of Operation** These refer to the financial costs of system operation which vary based on the number of services provided. Eg the cost of vehicle depreciation, drivers and energy use.
- **Timestep** The individual unit of time between updates of the simulation. This is taken to be one minute for all the evaluations performed in this thesis, but a different timestep could be used.
- **Headway** How long must the "gap" be between two vehicles travelling along a particular edge. Eg if the headway along an edge is two minutes, then after one vehicle starts travelling along an edge another vehicle must wait two minutes before it can start travelling along that edge. The headway of an edge is determined by how closely together vehicles can safely operate.
- **Intersection** A node served by multiple routes. A passenger wishing to travel between destinations not served by the same route will have to change at an intersection. Note that as most routes are bidirectional, almost all nodes are technically intersections (as they have both a forward and backward route). However we generally constrain the use of the term Intersection to refer only to where routes with different combinations of nodes and edges meet. These sort of nodes have special importance in many optimisation approaches, though in any of the ones we implemented in this report.

1.4 Limitations of the Problem to be Solved

1.4.1 Induced Demand and Demand Substitution

In reality, the public transport system of a city is impossible to disentangle completely from the overall functioning from a city. For example, higher quality public transport may cause passengers to switch from driving to catching public transport (demand substitution), it may even cause people to make trips that they would not otherwise have made (induced demand), much as improved air travel encouraged people to travel long distance much more than they did previously. Conversely worse public transport will have the opposite effect.

However these effects are extremely difficult to model and are really outside the domain of optimisation and into the domains of economics. Hence we will not consider them in this thesis. Instead we will make the assumption that passenger travel patterns, so the number of passengers trying to travel between any two nodes, is a constant regardless of the developed schedule(The route they take through the network will of course vary with the developed schedule). This should be a good enough approximation provided that all developed schedules are reasonably close to optimal, and the timescales being considered are short.

1.4.2 Physical Network Constraints

Another aspect of the problem which we must simplify is the issue of network constraints. The longer the time horizon, the less constraints there are on what can plausibly be optimised. For instance over the span of many years to decades, new pieces of transport infrastructure(eg new rail lines) can be developed. However attempting to optimise this is a very different and more complicated problem than simply optimising the level of service on existing infrastructure, and involves much more long term planning and arguably political decisions about the sort city of people want to live in. Hence it is beyond the scope of this thesis, and we will only consider the situation where physical infrastructure is fixed. For the same reason, optimisers implemented in thesis will only focus on the marginal cost of system operation. This is because fixed costs, like the cost of constructing rail lines and stations, do not vary noticeably with the amount of services using the network.

Conversely, in the short term not only is fixed infrastructure fixed, but there is also a maximum number of vehicles and staff available to operate on the network. However having a finite number of vehicles available turns the problem into the quadratic assignment problem, which is known to be NP hard [J and A 2002]. Furthermore new vehicles and staff can be purchased/trained much faster than infrastructure built, in a matter of a few months to at most a few years. Hence having no upper bound on the amount of vehicles allowed on the network at once (except that imposed by individual edges and nodes) will both make the problem more computationally tractable and also more relevant for medium term schedule optimisation(eg devising what next years train timetable will be). Another thing to note is that most public transport networks have far more demand during peak periods than outside the peak and hence will have plenty of idle vehicles during off-peak periods, meaning that for optimisation in the off-peak it will be unlikely to be necessary to worry about more vehicles being requested by the schedule than are actually available.

In this thesis we will mainly consider the problem of optimising the frequency of transport services along predefined routes, actually figuring out what the best routes are based on demand levels between nodes and available edges is a variant of the travelling salesman problem, which is known to be NP hard [J and A 2002] and hence computationally intractable. By using predefined routes we will hence dramatically reduce the required computational and software complexity. Additionally, for rail networks the routes a train can take are constrained by the track geometry, meaning that it is only practical for trains to travel along a small number of set routes anyway.

Additionally throughout this thesis we will be assuming that the amount of time taken to traverse a particular node is fixed and does not depend on the amount of vehicles using the edge. This assumption is useful as attempting optimisation where an edge can become congested makes the problem non-linear and hence harder to optimise. This is a good assumption for train networks, but more flawed for road networks where congestion is an issue. However for road networks, provided that public transport vehicles are a small fraction of total traffic they will not meaningfully affect congestion levels and hence the time taken to traverse an edge can still be assumed to be linear with demand(albeit likely to vary with the time of day as well as randomly between days based on random fluctuations in traffic levels).

1.4.3 Additional Constraints on optimisers

Another thing to keep in mind is certain types of optimisers may assume additional constraints to the system that the simulation does not assume. This is unsurprising because figuring out the best input to a complex system is generally much harder than actually getting the output from a particular input, hence optimisation requires more constraints to solve in a practical amount of time. We should design our simulation to be robust to a flawed timetable to prevent this causing any problems during schedule evaluation, though of course the results from these optimisers may be of lower quality.

1.5 Structure of this Thesis

Chapter 2 presents a literature review of existing work into the public transport schedule optimisation problem.

Chapter 3 presents an overview of the systems we intend to implement as part of our framework and the theoretical justifications for our approach. Chapter 3 also details the theoretical development of a basic optimisation approach of my own devising.

Chapter 4 then details how specific components of the framework were implemented in software. A user focused guide into operating the software is then provided in Chapter 5.

Chapter 6 then provides a demonstration of how the framework can be used, by using it too implement a slightly simplified version of the real world Sydney Trains system and trialing both the real world schedule and a schedule generated through the optimiser described in Chapter 3. The results of this experiment as well as conclusions and potential direction for future work are then discussed by Chapter 7. This concludes the main body of the thesis.

The thesis is then followed up by a bibliography which lists all resources used in the creation of this thesis. This is then followed by an appendix which includes code-listings for all software written for this thesis, as well as CSV files containing input data for the experimental comparison. Alternatively you can access the software and input data in my github repo at https://github.com/henryc47/Thesis_Public_Transport_Optimisation

CHAPTER 2

Literature review

2.1 Computer Modelling of Transport Systems

The development of a computer model of a public transport system is a key outcome of this thesis. Hence it is critical to consider the fundamental basis of how these models operate and what areas can effectively be simulated. Given that our model will need to operate on very large amounts of data and on very complex systems, it is also key to consider computer optimisation techniques for large scale simulations, to ensure that our model is able to run effectively with only finite amounts of computing resources and time to complete the project. Fortunately an extensive literature about this topic exists, and is discussed below.

2.1.1 Four Step Model of Transport

The four step model of transport has been widely used to model how passengers travel through a transportation network [McNally 2000]. It divides the transport process into four main stages.

- **Trip Generation**
  Calculates how many trips start and end at particular origins and destinations based on land use patterns, knowledge about economic geography, local attractions, etc

- **Trip Distribution**
  In this step, origins are matched with destinations to determine which trips are actually made.
- **Mode Choice**
  In this step, passengers choose which mode of transport they will use to travel through the network. Eg will they drive, catch public transport, walk, etc.
- **Route Generation**
  In this step, which route a passenger will use to move through the network is determined. Eg will they take one train line or the other to reach their destination. This then determines congestion levels on the network which then feedback and affect all other parts of the model.

The four step model was originally developed for road traffic by Marvin L Manheim [Manheim 1979], however it has also been applied with some success to public transport networks as well [Ahmed 2012].

The trip generation phase is the most complicated phase of the four-step model, requiring extensive information about local patterns of economic and social activity to produce an accurate model. This is well beyond the scope of this thesis. Instead we will utilise known origin and destination data about our transport network, which is publicly available information for many public transport systems. For instance, the origin and destination data for trips on the Sydney Trains network is available at [NSW 2018a].

The trip assignment phase is potentially quite useful for this thesis as while the number of journeys starting and ending at particular nodes is available for many transport networks, information of the form "a trip starts at a particular node and ends at another specific node" is rarely released for privacy reasons. The most widely used method of trip-assignment in the four-step model is the Gravity Model, which is discussed in it's own subsection

The mode generation and route generation phases are closely interlinked as the best mode of transport will depend on the best route for each particular mode of transport. At any rate in our model we will only be considering public transport, disregarding potential competition from private automobiles as that requires considerable additional work and would expand the scope of our thesis to an excessively large extent. Hence we will only use the route-generation phase of the four step model in our thesis. The route generation phase is an extremely critical phase as our simulation must ultimately simulate the flow of passengers through a public transport network, hence techniques to determine which routes passengers take are vital. These techniques are discussed further in the "Agent Pathfinding" Section.

2.1.1.1 The Gravity Model

The gravity model of trip assignment is widely used as part of the four-step model [McNally 2000]. In the gravity model, the likelihood in which a passenger starting at a particular origin node travels to a particular destination node is based on a combination of the distance between the nodes and the total number of passengers leaving at that destination node. A passenger is considered to be more likely to be travelling to a nearby node and also more likely to be travelling to a node which many other passengers are also travelling towards.

2.1.2 Agent Based Modelling

Agent based modelling is a key technique used for transport network simulation. In agent based modelling, individual agents are autonomous and follow a set of common rules while attempting to achieve their own private goals within the simulation, interacting with other agents and the environment in the process [Bonabeau 2002]

In the public transport context, agents represent passengers. The goal of each agent is to travel from their origin to their destination in the fastest way possible.

Agent based models offer a key advantage over models based on aggregate demand flows as they more accurately reflect the behaviour of real individuals, and hence are better able to accurately simulate changes in network structure [Kagho et al. 2020]. They are also more straightforward models to understand and implement. Unfortunately the greater detail and granularity of agent-based models comes at a cost. The need to simulate every single passenger on the network requires considerable computing resources. However in the modern age we now have access to extremely large amounts of computing power, allowing for simulations with potentially millions of agents to be run in practical amounts of time [Parry 2009].

This means that we are now able to practically simulate a good-approximation of a large scale urban public transport system, with millions of passengers each day, inside a computer model. However it also means that optimisation techniques will be critical to building a practical model. These are discussed further in the "Agent Pathfinding" section of the literature review, as well as in the "Methodology" section.

2.1.3 Agent Pathfinding

Passengers on any sort of transport network wish to travel in the fastest (or perhaps the cheapest manner). To be accurate, any agent-based path-finding simulation will need to include an algorithm able to calculate the lowest cost path(which may not necessary be the lowest financial cost, it could be the time taken) which the agents will try and take between their origin and their destination. In addition to producing the lowest case path for passengers, for system practicality it is also essential to use an algorithm which is computationally efficient as it is path-finding that is the major bottleneck in an agent based simulation [Strandberg et al. 2016].

Numerous high-quality algorithms for general path-finding through a network exist, with the most notable and widely used being Dijkstra's algorithm [Dijkstra 1959] and it's derivatives, the most notable of which is [A* Doran and Michie 1966]. Dijkstra's algorithm uses an exhaustive best first search technique to find the shortest path from a node to all other nodes(perhaps terminated early if a particular destination node is reached), while A* uses a heuristic to try and focus it's search effort on a particular destination node.

Both Dijkstra's and A* are guaranteed to calculate the best path-finding solution provided that the network has no negative cost nodes and in the A* case that the heuristic is consistent and admissible, which in layman's terms means that the heuristic never overestimates the cost to reach the goal. However unfortunately these two algorithms can be can quite slow on large networks, with even a well optimised implementation of Dijkstra's algorithm having a worst case time complexity of $O((n + e) \log n)$, where n is the number of nodes and e the number of edges. [Fredman and Tarjan 1984]. This may well prove to be a problem in our simulation as real public transport systems can have hundreds of nodes and edges, and millions of agents performing path-finding.

Hence to develop a practical simulation, performing additional optimisation on the performing algorithm may be key. A key method for this which has showed considerable promise are contraction hierarchies [Geisberger et al. 2008]. This method "ranks" the nodes by importance and then "contracts" the network by generating "shortcuts" between the more important nodes, where the shortcuts are simply the sum of the edge weights on the shortest path between those two nodes. These shortcuts can considerably speedup the path-finding time on large complex networks. This method is normally applied to road networks and takes advantage of the fact that road networks tend to be very hierarchical, with less important local streets almost never being the fastest path for long-distance traffic. In public transport networks, there tends to be a much less clear hierarchy of routes however there is still generally a clear hierarchy of nodes, as only a small proportion of nodes actually provide an intersection between routes in a typical public transport system.

It should also be noted that some success has been had applying contraction hierarchies to situations where edges costs are not fixed [Dibbelt et al. 2014]. This is a comparable scenario to the public transport scenarios, where the edge costs are variable due to variable weighting times for a service depending on when a passenger arrives at a node.

There are also some minor differences between the sorts of networks Dijkstra derived algorithms are designed to solved and the public transport system we are attempting to model. Most notably, passengers are normally assumed to be able to travel along an edge at any time. This is an accurate reflection of private travel networks, however in public transport networks they must wait for the arrival of a vehicle travelling towards their destination. Techniques to resolve this problem are discussed in later chapters of this thesis.

2.2 Schedule Optimisers for Public Transport

While this thesis primary focus is on developing a set of software tools to evaluate schedule optimisers, implementing a variety of schedule optimisers is still a key component of this thesis. While we will not be focusing on developing new optimisation techniques, implementing existing optimisers and making slight optimisers is still useful. This is principally because implementing existing optimisers will allow us to generate the schedules which the main simulation needs to run. This serves to validate the simulation and our conception of the network constraints. Additionally, implementing existing optimisers allows us to see how well these optimisations strategies perform on much larger and more complex problems than they were initially tested on, as most transport optimisation papers only evaluate their strategies on small-scale and very simplified transport networks. Testing on our more complex model will be highly useful as the complexity will be much more comparable to real urban transport networks.

An extensive literature about potential strategies for optimisation exists and we have discussed key parts of it below. The key things to keep in mind is that the public-transport scheduling problem is non-convex and so approximate solutions are necessary to produce results in reasonable amounts of time.

2.2.1 Schedule Synchronisation Problem

The schedule synchronisation problem (SSP) tries to minimise the time passengers spend waiting at an interchange between services. Given that many trips on a large public transport network will require interchanging from one route to another at an intersection, this is a critical problem to solve when developing a good quality schedule optimiser. Noted in [J and A 2002] to be NP hard in it's pure form, it nonetheless has been solved approximately be other authors. For example using a Tabu search [S 1992]. Of course algorithms to solve the SSP problem only optimise for minimising waiting period at an interchange, and it may be necessary to combine it with other algorithms to develop an optimiser for a whole transport system.

2.2.2 Tabu Search

Tabu search [Glover 1986] is a modified form of greedy local search. Like in local greedy search algorithms, the optimisers looks at solutions similar to the starting solution and checks nearby solutions in the hope of spotting an improvement. However unlike in pure greedy search, the optimiser will if there are no nearby better solutions, consider nearby solutions which are worse than the current solution in the hope that some that solutions neighbours might offer an improvement. This makes tabu search useful for non-convex optimisation problems such as the public transport optimisation problem[S 1992]

2.2.3 Simulated Annealing

Simulated annealing [Pincus 1970]is an optimisation method similar conceptually to Tabu search in that it finds and approximate solution to a non-convex problem through it's willingness to consider nearby less optimal solutions in the hope that it will uncover new regions of greater optimally. In an analogy with real-world annealing, early on in the simulated annealing process the willingness for the solution searcher to move in non locally optimal directions in the solution space is high, but this reduces over time as the algorithm hopefully settles near the global minimum. It has been applied successful for the problem of scheduling intercity buses [Rodriguez et al. 2014] and of designing optimal bus routes through a city [Fan and Machemehl 2006]

2.2.4 Genetic Algorithms

Genetic algorithms are a form of machine learning which an evolutionary process akin to natural selection to solve a system. Essentially numerous random solutions are selected, the solutions are then tested and evaluated for quality, the best solutions from the first generation are kept and mutated to produce the second generation. This process is repeated until a sufficiently good quality solution is obtained. Genetic algorithms are a potentially useful approach to many types of problems as they are able to solve many problems, even ones which are very hard to understand explicitly (eg In the real world they have managed to evolve life to live in many hostile environments, and solve many problems like walking and sight). As such it is unsurprising that many authors have also had success in using them for transport optimisation. For example [P 2003] has used them to solve the SSP problem, while [Șerban 2021] used them to solve the issue of optimal line frequency(how frequently should services arrive on a given route).

Unfortunately genetic algorithms can be difficult to implement as it is conceptually quite difficult to formulate a genetic algorithm able to generate schedules in a semi-random way without violating the physical network constraints. [Șerban 2021] provides some interesting approaches to how to formulate the problem of evolutionary schedule generation using the concept of "chromosomes"

Genetic algorithms also have the downside that the evaluation of the fitness function is extremely computationally intensive as the full model of the transport system must be simulated. This is particularly a problem for large complex networks like those we intend to model in this project.

CHAPTER 3

Overview of Framework

3.1 Overview of Developed Framework

The high level structure of our project consists of four major software modules. These are summarised below. The high-level relationships between the modules are depicted in figure 3.1. Data is represented as rectangles and software modules as circles.

- A simulator which is able to simulate the provided public transport system. The simulator contains a representation of the public transport network.
- An optimiser which can generate an optimised schedule for that public transport system.
- An evaluator which can use the output of the simulation to determine summary statistics allowing the performance of a schedule (whether manually written or optimiser generated) to be evaluated
- A graphical user interface (GUI) which makes it easy for the end user to configure the system for simulation and visualise the results.

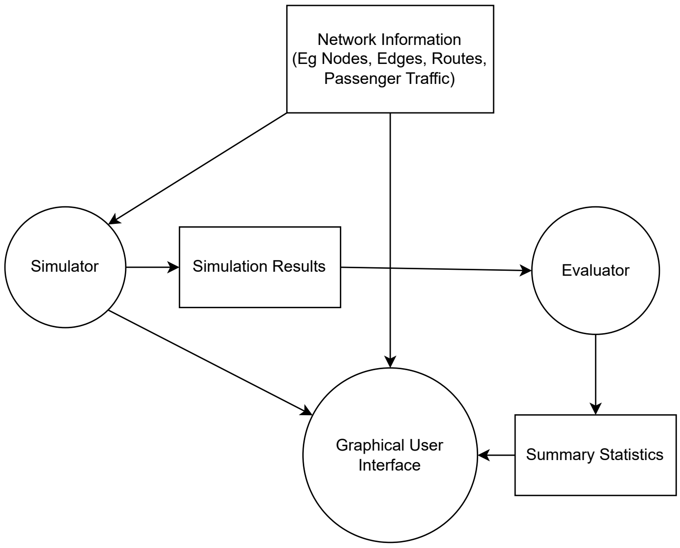


FIGURE 3.1: Relationship between Software Modules

3.1.1 Simulator

3.1.1.1 Network Representation

The simulator contains a representation of the network. This representation consists of a graph consisting of nodes and edges, which represent for instance train stations and the lines which connect them. An example of such a graph is included in figure 3.2. Vehicles and the agents they carry are able to travel down the edges in a set amount of time which is a property of the edge.

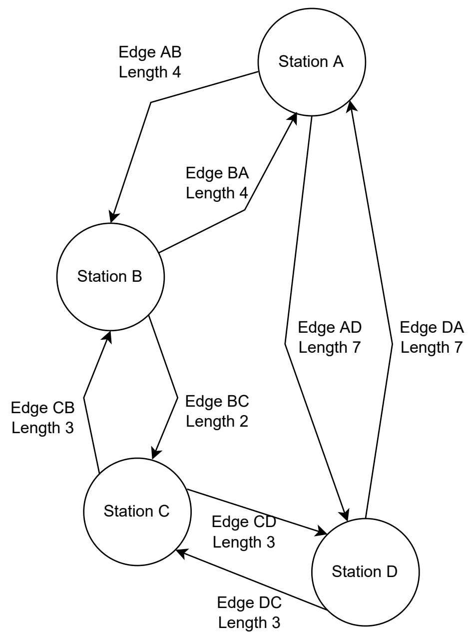


FIGURE 3.2: An example of nodes and edges

3.1.1.2 Agents

Agents represent the passengers travelling through the network. Each agent is created at an origin node with the goal of reaching a destination node. As agents can only travel along an edge inside a vehicle, they must calculate the optimal combination of services to reach their destination, using the schedule to determine how vehicles travel through the network. This is a process called path-finding and is discussed in detail in it's own section. A flow-chart representing the life of an agent is shown in figure 3.3.

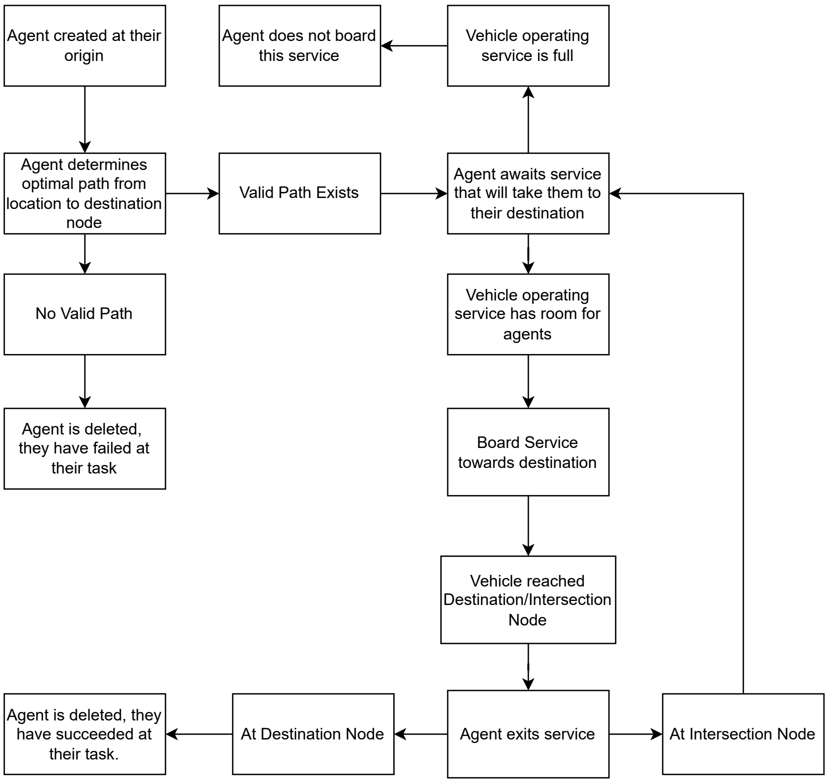


FIGURE 3.3: The life of an agent

As noted in the flow-chart, there may sometimes not be a valid path between the origin and destination node. In this scenario, the agent is deleted and a penalty applied to the evaluation function to discourage optimisers from allowing this situation from happening frequently.

3.1.1.3 Vehicles

Vehicles travel around the network, transporting agents from their origin to their destination. Vehicles move along edges between nodes following their specified route in a process known as a service. The time when vehicles following a particular route are dispatched to provide a service is controlled by the network schedule, discussed in the next section. A diagram representing the life of a vehicle is shown in figure 3.4.

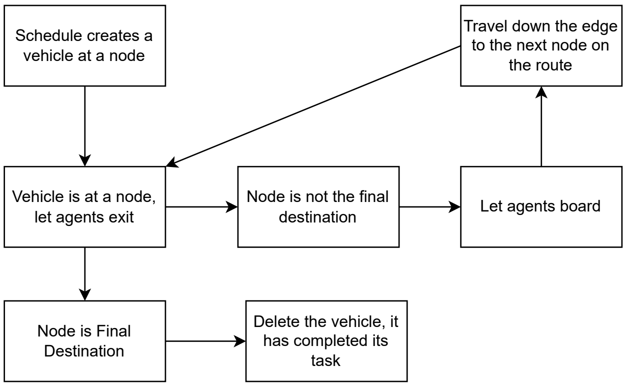


FIGURE 3.4: The life of a vehicle

3.1.2 Routes and Schedule

A route is a combination of nodes and edges through which a vehicle will travel as part of a service. The schedule or timetable determines when vehicles providing each services will be created. The schedule may be either manually generated or generated through an optimiser.

3.1.2.1 Simulation Process

The simulation process implements a modified form of the four-step model [Manheim 1979] of trip generation and distribution. In our case the third mode split is neglected as First passengers are generated at nodes. The number of passengers generated at each node is controlled by the provided passenger behaviour statistics.

The determination of which nodes passengers wish to travel too is then determined using the Gravity Model. The gravity model is used because generally only station entries/exits are provided, how individual passengers travel through the network is rarely recorded for privacy. Hence we must have a method of estimating how individual passengers travel so the simulation agents can replicate the behaviour. However if full origin-destination travel data is available to the end-user, it would be straightforward to modify the programme to use this data instead of the Gravity Model.

Lastly path-finding is performed to determine how agents travel through the network. This is discussed in more detail in the next section.

This process is implemented in the OOP paradigm as a series of steps, which occur at every time-step. At each time-step, all objects of a certain class performs the relevant action. This process is depicted in figure 3.5.

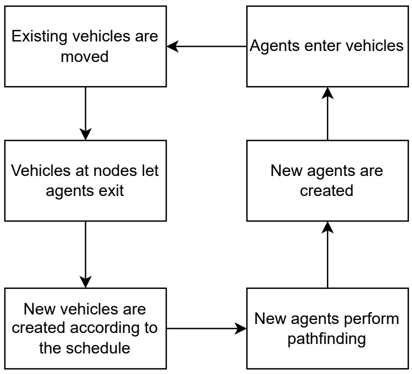


FIGURE 3.5: Simulation Main Loop

3.1.2.2 Pathfinding

Finding the shortest route between an agents origin and destination node is a key part of the simulation as it allows agents to decide which services to catch to reach their destination.

Conventional path-finding algorithms such as Dijistrakas and A* assume that an agent is allowed to travel along any edge at any time, while this is an accurate assumption for travel by private car, this is not true for public transport, where agents can only travel along an edge with a vehicle whose route includes that edge. However if rather than the edges of the graph being the physical infrastructure of the network, we consider the edges of the graph to be a vehicle travelling between two nodes at a specific time, we can still use conventional path-finding algorithms. An example of this new representation of path-finding edges is included in figure 3.6. Implementation details of path-finding algorithms used is included in the next chapter.

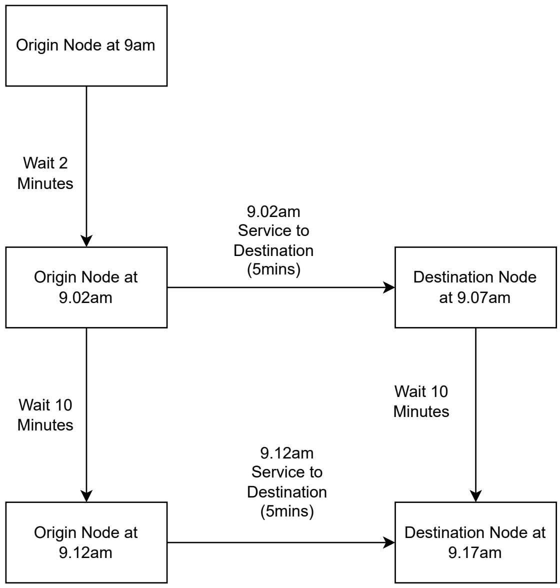


FIGURE 3.6: Representation of Public Transport Edges for Pathfinding

3.1.2.3 Output

The simulation process logs all relevant data regarding the position of vehicles and agents and their movement through the network, to enable the evaluator to produce relevant conclusions from the simulation and to allow the GUI to display graphically the simulation results.

3.1.3 Optimiser

The optimiser will utilise provided information about the network, including passenger data and associated costs of passenger time/vehicle operation to determine the optimal time for services along particular routes to be dispatched. The optimiser used in the evaluation of this framework, discussed further under the **Henry Convex** chapter, only varies the frequency along predefined routes to try and balance system cost with passenger wait times. However more sophisticated optimisers could be implemented to achieve a more sophisticated schedule

3.1.4 Evaluator

The evaluator will utilise the collected statistics from the Simulation to produce a report highlighting key results of the simulation. The specifics are described in the Implemention chapter of this report.

3.1.5 Graphical User Interface

The graphical user interface (GUI) acts a glue binding the other components of the framework together. The GUI provides controls to import files, setup the simulation and select an optimiser to use. It also implements extensive visualisation tools to visualise both network details (eg passenger trip distribution, position of nodes and edges) as well as the simulation, enabling the end user to see how passengers and vehicles move through the network over time. Further details are discussed in Chapter 4 and 5.

3.1.6 Required Data

We must represent all details of the proposed network and schedule in a standardised format which makes it easy for the end user to import the details of their chosen network into the framework. We must be able to represent.

- Properties of all nodes in the system, eg geographic location
- Properties of all edges in the system, eg travel time, which nodes are linked
- Properties of all vehicles in the system, eg max passenger capacity
- Possible routes which vehicles can take through a network
- For a preset schedule, the frequency at which vehicles travel along those routes.
- Passenger behaviour statistics, how many passengers want to travel between each node pair.

Further details of how this is implemented is included in the next chapter.

3.2 Software Design Philosophy

To effectively implement our framework in software, we must decide on a software design approach that most effectively allows us to achieve our goals of building a robust framework that simultaneously offers good performance as well as being robust and easily extensible and adaptable. Our software developed should hence be modular in nature, allowing for modification to parts of the program without requiring alterations to other components.

To achieve this, we implemented each of the previously discussed software modules using a combination of the Functional and Object-Orientated Programming (OOP) Philosophy.

Functional programming was used for smaller subroutines as it is straightforward to write and understand smaller segments of code using the functional framework.

OOP was used for the larger scale modules. This was done using the Facade Pattern of OOP Design [Gamma et al. 1994]. In this software pattern each major module the program has an interface to other parts of the program (in our case the standard data-formats used to communicate with other parts of the program). This greatly assists in development as it allows for one module to be changed without requiring changes in other modules, provided the interface (import and export data) is not changed. This also makes it much easier to extend the software. The schedule optimiser is also an example of the Adaptor Pattern [Gamma et al. 1994], as the schedule optimiser presents a common interface for a wide variety of different potential optimisers.

OOP was also used for the implementation of the simulation. Each instance of a Node, Edgee, Agent, Vehicles or Schedule are objects of that class which interact with each other as they play their provided role in the network. Representing each simulated instance as a separate object makes it straightforward to vary the parameters of each instance (eg which node does an agent need to path too) without affecting the whole class.

3.3 Language and Libraries Used

We implemented the software using Python 3 [Foundation 2022] due to it being open-source, easy to use for both OOP and functional programming, as well as having a wealth of open-source modules and or libraries implementing everything from path-finding algorithms, statistical analysis tools and graphical user interfaces. This made the framework much easier to develop and will also make it much easier to modify and extend for future users.

The downside of Python is that it has relatively poor performance compared to lower level languages such as C++. However we felt that for initial project development, ease of development outweighed performance concerns. This assumption proved correct as in the end a full simulation of a days operation of the Sydney Trains network only took about three minutes to run on our hardware, which is sufficiently fast performance for our purposes. Furthermore if performance were to become an issue in some future application (for instance if very large networks were to be simulated, or an optimiser were to be used which needed to run many simulation iterations), Python does make it relatively straightforward to use libraries written in C/C++ to gain much faster performance. This combined with the modular design of our framework should make it easy to reimplement performance critical areas, such as pathfinding, in C/C++.

To maximise ease of use and robustness, CSV files were selected as the primary method in which external data would be imported into the programme. CSV files are a widely used format which is widely used, human readable and easy to edit using standard spreadsheet software.

In addition to the python core libraries. We used the following open source libraries for the development of this thesis.

- **NUMPY** Developers 2022 was used for large scale mathematical calculations, particularly those in the gravity model.
- **PANDAS** NumFocus 2022 was used for importing CSV files into the program.
- **Tkinter**, an integral part of Python, was used for the implementation of the GUI.

3.4 The Henry Convex Optimiser

To test our framework, we developed a basic optimisation approach which attempts to solve the optimal frequency problem of a public transport service using convex optimisation. This requires some simplification assumptions about the problem to be solved, however as shown in Chapter 7 in proved capable of producing decent results on a complex system. The theoretical basis for this optimiser is discussed below.

Consider a public transport system consisting of only one route between a single origin and destination, which costs k currency per service provided and takes a hours. Suppose that passengers arrive continuously at a constant rate of n passengers/hour and catch the next arriving service, which arrive every w hours. Let's also suppose that passengers time is valued time at v currency/hour. Note with constant service frequency average wait time is $w/2$.

We can add the total cost of providing the service(per hour) k/w to the value of the passenger time lost in transit, $(\frac{w}{2} + a) * nv$. This give us the total cost(compared to an ideal scenario where passengers freely teleport from origin to destination) of

$$C = (\frac{1}{2}w + a)nv + \frac{k}{w}$$

This is a convex function which can be minimised to find the optimal wait time.

$$\frac{dC}{dw} = (\frac{1}{2}nv - k\frac{1}{w^2}$$

As k,w $> 0 \therefore \frac{d^2C}{dw^2} = k\frac{1}{w^3} > 0$, and the function is convex, hence to minimise

$$\frac{dC}{dw} = 0 \therefore \frac{1}{2}nv = \frac{k}{w^2} \therefore w = \sqrt{\frac{2k}{nv}}$$

This simple equation above can be used to find the optimal wait time. As it uses Convex optimisation and was developed by the author whose name is Henry, we will refer to it as the Henry Convex Approach. We can easily calculate the cost of a route by multiplying the time taken to traverse the route by the assumed cost of operating a vehicle.

In a complex system like the Sydney Trains system we are modelling, there will often be more than one route passing through a node. As the number of passengers travelling along a particular route from that node will be lower than the total number arriving at that node, the optimal frequency along each route will be lower than the number of total passengers would indiciate.

We accounted for this in our software implementation by dividing the amount of passengers considered at a node by the number of routes passing through that node. Hence nodes which have many routes passing through them will have less influence on the optimal wait time. We implemented this in the function **Henry_Convex**.

CHAPTER 4

Implementation of Framework

4.1 Importing Data - CSV Descriptions

Importing data in the framework is done through CSV files. A description of the required CSV files is included below. Example CSV files are also available in appendix B and at my github at `https://github.com/henryc47/Thesis_Public_Transport_Optimisation`

4.1.1 Nodes CSV

The Nodes CSV file provides information about the nodes in the network. Each node has it's own row, with each column storing particular information about each node. These columns are as follows.

- **Name** - This is the name of the node
- **Daily Passengers** - This is the number of passengers who start their journey at this node.
- **Location** - This is the geographic position of the node in the format Latitude,Longitude

4.1.2 Edges CSV

The Edges CSV file provides information about the edges in the network. Each edge has it's own row, with each column storing particular information about each edge. These columns are as follows.

- **Start** - This is the name of the starting node of the edge
- **End** - This is the name of the ending node of the edge
- **Time** - This is the time taken in timesteps (minutes in our scenario) to traverse the edge.
- **Bidirectional** - Can the edge be traversed in both directions? If set to yes, during the setup of the simulation an identical edge will be setup with the start and end node reversed.

4.1.3 Schedule Segments CSV

The Schedule Segments CSV file provides information about how multiple edges can be connected together to provide a service. It can be used to build up more complex schedules either manually or with a more sophisticated optimiser automatically. Each segment has it's own row, each column column storing particular information about each segment. These columns are as follows.

- **Name** - This is the name of the segment in the format StartNode-EndNode. This format is required so that the segment can be reversed to provide a segment in the opposite direction.
- **Modifier** - This is an addition to the name of the segment which allows differentiation between two segments with the same starting and ending node, eg using a different route or an express service which skips some nodes.

- **Schedule** - This is a comma separated list of all the nodes names in the segment in the order from StartNode-EndNode. To generate the reverse segment the list can be reversed. During the simulation setup stage, this node names in this list are used to determine the edges which make up the schedule.

4.1.4 Schedule CSV

The Schedule CSV file provides information about the schedule of the the network. Each route has it's own row, with each column storing particular information about that route. The columns are as follows.

- **Name** - The name of the route
- **Gap** - The gap in timesteps (minutes in our scenario) between services along this route.
- **Offset** - The time in timesteps between the beginning of the simulation and the first service of this route.
- **Finish** - The time in timesteps after after which no more services of this route will be generated.
- **Schedule Segments** - The list of schedule segments in comma separated form which make up the route.

Note that all this information is only required when using manual scheduling (indeed this is the manual schedule). The Henry Convex Optimiser sets its own Gap to a calculated optimal value for each route, while more advanced optimisers could set their own offset and finish times as well. An optimiser which can determine it's own routes could dispense with this file entirely, determining routes from schedule segments or even individual edges.

4.2 Evaluation CSV

The Evaluation CSV contains costs common to the CSV file and is used by the Evaluator to convert from numbers of passengers and vehicles into costs. Hence there is only one data-row. The columns are as follows.

- **Vehicle Cost** - The marginal cost of operating a vehicle in dollars per hour
- **Agent Cost Seated** - The opportunity cost or dis-utility of a passenger being seated inside a vehicle for an hour.
- **Agent Cost Standing** - The opportunity cost or dis-utility of a passenger standing inside a vehicle for an hour
- **Agent Cost Waiting** - The opportunity cost or dis-utility of a passenger waiting for a service for an hour.
- **Unfinished Penalty** - Penalty in dollars for dis-utility of a passenger not being unable to make their journey.

4.3 Parameters CSV

The Parameters CSV File contains parameters common to the whole simulation. Hence there is only one data-row. The columns are as follows.

- **Vehicle Max Seated** - The maximum of passengers who can be seated in a vehicle
- **Vehicle Max Standing** - The maximum number of passengers who can fit in a vehicle (both standing + seated capacity)
- **Traffic Time Gap** - The amount of timesteps represented by rows in the Scenario CSV file.

4.4 Scenario CSV

The scenario CSV file represents how traffic varies throughout the day. It contains one column, **Traffic Multiplier**. This represents how the total volume of traffic varies over the course of the day as a multiple of the total daily traffic that occurs per hour during that row. In our scenario, each row corresponds to one hour.

4.5 Simulator Implementation

We implemented the network simulation in software inside the **Network Class** in the **network.py** file. The network consists of edges and nodes which are instances of the **Edge** and **Node** classes respectively, these are also in the **network.py** file. Each instance of a Node stores a reference to the edges it is connected too, and each Edge stores a reference to the node object it is connected too. At the start of the simulation, the network of nodes and edges is constructed from the list of nodes and edges using name matching. Hence each node must have a unique name and each edge must have a unique pair of origin and destination nodes (the combination must be unique, one node can have multiple edges)

The **Agent Class** in **agent.py** was used to simulate agents/passengers, the **Vehicle Class** in **vehicle.py** was used to simulate vehicles. As the agent class travels through the network, a reference too it is stored inside the Node or Vehicle is currently at, allowing for easy counting and logging of the number of passengers at a node. Note for computational efficiency reasons, one Agent represents all the passengers travelling between a particular node pair at a particular time-step.

The **Schedule Class** in **schedule.py** represents the route a vehicle follows as a series of references to nodes and edges. As the vehicle travels through it's schedule, it communicate with the Node and Agent objects when it is at a Node to allow Agents to board and alight where required.

A matrix of the total traffic travelling between every node pair, generated by the Gravity Model as discussed earlier and stored as a variable in the network class. is used to create Agents as required.If the calculated number of a passengers to be created to travel between a specific origin and destination at a specific time-step is less than one, we setup the system so that there will be a random probability of a single passenger agent being created equal to the calculated number of passengers. The same approach is used for the non-integer component of the calculated number of passengers if the calculated number of passengers is more than 1. The times at which vehicles of particular routes are to be dispatched is calculated at simulation start based on the frequency information for each route either calculated by the optimiser or provided manually as part of a predetermined schedule.

The update process described in the previous chapter is performed every time-step to update the simulation results, until the user specified end of simulation time-step is reached.

4.5.1 Gravity Model Implementation

Information about patronage levels at particular stations over time is widely available. For example Transport for NSW publishes how many passengers entered/exited particular train stations at particular times [NSW 2018a]. Unfortunately actual origin/destination of individual passengers data is rarely publicly available due to privacy concerns.

As our program requires origin/destination pair passenger trip data, we synthesised origin/destination passenger trip data using the Gravity Model as discussed in McNally 2000 However we expect that we will be able to synthesize decent quality approximations of origin/destination movements using the Gravity Method [McNally 2000]. We implemented this in software using the function **gravity_assignment** in the **network.py** file. We found that the gravity model is not numerically stable, and the total number of passengers starting from or going to a node is different after applying it than the original number of passengers. To remove this error, we used an iterative approach where the weights of nodes in the gravity model were scaled down if too many passengers were estimated to be using that node, or scaled up if there were too few. After a moderate number of iterations, we were able to get the difference to be negligible. We set our function to cease iterating after the gravity model reached within plus or minus 0.1 % of the true value.

Once the origin destination passenger data has been generated through the gravity model (or if available provided directly), it is stored in the **Network** object, awaiting the running of the simulation

4.5.2 Pathfinding Implementation

Pathfinding is the computationally intensive part of the framework and hence finding an efficient pathfinding algorithm is key

We initially used our own custom variant of the A* algorithm to perform path-finding. This variant used the best case scenario travel time (the time it would take to travel between two nodes if you never had to wait for a vehicle) as the heuristic, serving as a suitable lower bound to focus the search algorithm given that the traditional euclidean distance heuristic does not make sense in a scenario where you are trying to minimise travel time. However we discovered that this algorithm provided very poor performance, with a simulation of the full network for a day taking over an hour (exact results were not recorded). We noted that even with the direction provided by the A* heuristic, much of the network was still being searched for every single agent generated. Given that we generated many agents for each node at each timestep (usually dozens, and up to hundreds at the more important stations), it would likely be much more efficient to use Dijistraka algorithm, which finds a path from a node to all other nodes in the network, allowing it to be used to find the path for all agents generated at a node simultaneously. This dramatically cut down on the simulation time.

We further improved our implementation of Dijkstra's by only ending the search once all destination nodes from a node had been found. As most of the network would still need to be searched in most cases, this only provided a minor speedup, but the speedup from less path-finding still outweighed the additional overhead of checking whether the destination had been reached.

We realised that for a particular node, as vehicles do not arrive every timestep the possible paths an agent can take too it's destination do not change every time-step. Hence it is a waste of time to update the path-finding for every node every time-step. We changed our code to only update the path-finding for a node when a vehicle travels through that node, and we noticed an approximately two-fold speedup in the simulation, which is roughly what you would expect given that most nodes have a vehicle arriving every few minutes and most of the program time is spent on pathfinding.

4.6 Optimiser

For programming simplicity, we implemented the **Henry Convex** and **Henry Convex Simple** optimisers inside the **Network** class as the methods **Henry_Convex_Optimiser** and **Henry_Convex_Optimiser_Simple**. Long term it would be better to move the optimisers into a separate object and file, to enable more alteration and comparison of optimisers.

4.7 Evaluator

The evaluator is a very simple program contained in **Evalator.py** which takes in the data recorded during the simulation and combines it with the cost assumptions made in the evaluation CSV to determine the passenger cost and financial cost of the network. It produces a text report which provides summary statistics of the simulation. These are as follows

- Number of passenger trips
- % of trips reached their destination
- Total time per passenger in total
- Total time per passenger standing
- Total time per passenger sitting
- Total time per passenger in total
- Cost of operating vehicles in the network
- Max number of vehicles at once
- Max passengers in a vehicle
- Combined vehicle operation and passenger time cost
- Vehicle operation cost per passenger
- Vehicle+Time cost per passenger

4.8 Graphical User Interface

The evaluation engine as described in the overview is not just the report generating component, but also the whole interface which allows the end user to control the program. The graphical user interface (GUI) implements extensive visualisation tools to visualise both network details (eg passenger trip distribution, position of nodes and edges) as well as the simulation, enabling the end user to see how passengers and vehicles move through the network over time. The GUI also provides controls to import files, setup the simulation and select an optimiser to use. The GUI was written using Tkinter, Pythons inbuilt GUI library.The implementation can be found in **render.py**. As this is not a thesis on GUI design, it's internal workings will not be discussed further. However an extensive user manual is provided in Chapter 5.

CHAPTER 5

User Guide

5.1 Program Launching

Operation of the framework is done using a graphical user interface. To start the program, open up a python terminal in the same folder as the project code. This project code can be obtained from `https://github.com/henryc47/Thesis_Public_Transport_Optimisation`. It is also available in the appendix.

Inside the computers terminal, run the command **Python main.py**. If you have Python 2 installed in addition to Python 3, run **Python3 main.py**. This runs a setup script which launches the graphical user interface. This should cause a window to pop up on your screen. This graphical user interface (GUI) can be used to configure the simulation and view the results. The GUI can been seen in figure 5.1. The nodes are represented as circles, the edges correspond to lines. Node and edge locations are accurate to the real world Sydney Trains Network.

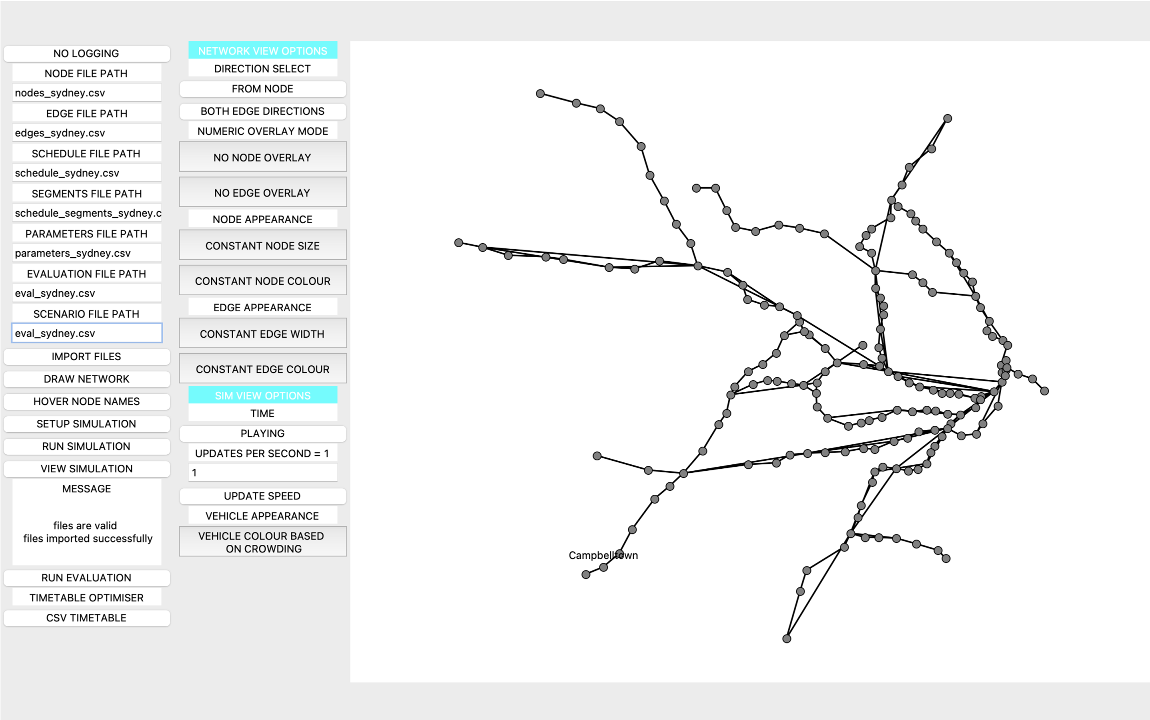


FIGURE 5.1: The GUI Displaying the Sydney Trains Network

5.2 Cross Platform Compatibility

This program was design and developed for the MacOS operating system. While I have endeavoured to use only cross-platform libraries and the software should work on any platform with a functioning Python interpreter, operation on non MacOS platforms is not certain. While the core simulation and evaluation codebase should work identically across platform, some aspects of the GUI are likey to vary cross-platform or require minor alterations to work effectively.

5.3 Operations Guide

Upon creation, the GUI will display a selection of widgets which can be used to setup and run the simulation. Their form and function is as follows

- The Logging Button, which initially says "NO LOGGING", can be clicked to alternative through different logging levels. Depending on the logging level, different amounts of warning and information will be printed to the console during system operation
- The **NODE FILE PATH** textbox should be used to enter the relative filepath to the NODES CSV file used to store information about the nodes in the network.
- The **EDGE FILE PATH** textbox should be used to enter the relative filepath to the EDGES CSV file used to store information about the edges that make up the network.
- The **SEGMENTS FILE PATH** textbox should be used to enter the relative filepath to the SCHEDULE SEGMENTS CSV file used to store information about the segments that will be used to make up the schedule
- The **SCHEDULE FILE PATH** textbox should be used to enter the relative filepath to the SCHEDULE CSV file used to store information about the schedule that will be used to simulate the network
- The **EVALUATION FILE PATH** textbox should be used to enter the relative filepath to the EVALUATION CSV used to store information about the costs used by the evaluator and optimiser.
- The **PARAMETER FILE PATH** textbox should be used to enter the relative filepath to the SIMULATION CSV file used to store information about the simulation parameters.

- The **SCENARIO FILE PATH** textbox should be used to enter the relative filepath to the SCENARIO CSV file used to store information about passenger volumes vary through the simulation.
- All file path widgets are prefilled with the provided example CSV files for convenience.
- the **IMPORT FILES** button can be clicked to import the files selected in the preceding widgets.
- the **DRAW NETWORK** button can be clicked to draw the network represented described in the node and edge files on the GUI.
- the **HOVER NODE NAMES** button can be clicked to toggle between only displaying the names of nodes in the nodes network when the mouse hovers over a node, and displaying the names of all the nodes at once.
- the **SETUP SIMULATION BUTTON** button can be clicked to perform various tasks that need to be completed before a simulation can be run, most notably determined levels of passenger traffic between node pairs using the gravity model. Depending on the size of the network, this may take several seconds.
- the **RUN SIMULATION** button can be clicked to run the simulation once it is setup. Depending on your setup this may take several seconds to many minutes.
- the **VIEW SIMULATION** button can be clicked to view the simulation once it has been run.
- the button below **TIMETABLE OPTIMISER** can be clicked to toggle between optimisers. The initial selection is **CSV TIMETABLE** in which frequency is set by the SCHEDULE CSV file. Other options available are **HENRY CONVEX** and **HENRY CONVEX SIMPLE** which implement the optimisers described by earlier. The optimiser must be selected before simulation setup is performed.

To setup and run the simulation, the user should first input the file-path to the file containing the information about nodes, edges and schedules. They should first click the **IMPORT**

FILES. If we wish to change the Optimiser using the **TIMETABLE OPTIMISER** button they should do so. They should then press the **SETUP SIMULATION** and **RUN SIMULATION** buttons in that order to setup and run the simulation using the provided files. Note the setup simulation step will automatically draw the simulated network on the provided canvas.

Once the simulation is setup and run, the GUI includes many options to configure the display of the results. This can be done in the menus under the NETWORK VIEW CONTROLS and textbfSIMULATION VIEW CONTROLS menus.

Regardless of the options selected, a map of the network will be displayed in the GUI. The user can use the mouse to move around the network by dragging, they can also zoom in and out using the scroll wheel. This is very useful in larger more complex networks where the detail can be hard to see when the whole network is visible.

The **NETWORK VIEW CONTROLS** has many options to configure the network display. These include toggle buttons to control the size and colour of nodes and edges, as well as to display relevant numeric information about nodes. The size and colour of nodes can be set based on the expected number of passengers (from the gravity model) going too/from a node either to another node or too all other destinations, as well as the actual number of passengers waiting at a node while the simulation is being run. Similarly the size and colour of edges can be based on predicted traffic level from the gravity model as well as actual traffic levels from the simulation. It is also possible to use the length of edges and the travel time between nodes to control the size and colour.

Once the simulation has been run the **VIEW SIMULATION** button can be used to start playback of the simulation. Little squares representing vehicles will be seen to move around the network. The **SIMULATION VIEW CONTROLS** has controls to pause/start the playback of the simulation, alter the playback speed and also control the appearance of the vehicle. The size and colour of vehicles can be set to vary with the level of crowding of the vehicle. Mousing over a vehicle will display it's origin and destination and the number of passengers currently onboard it. The **SIMULATION VIEW CONTROLS** section also includes a clock which informs the user what the current time being displayed is. Note the colour scale runs from blue-green-yellow-red as the quantity increases.

Some demonstrations of the GUI are displayed in figures 5.2 to 5.4.

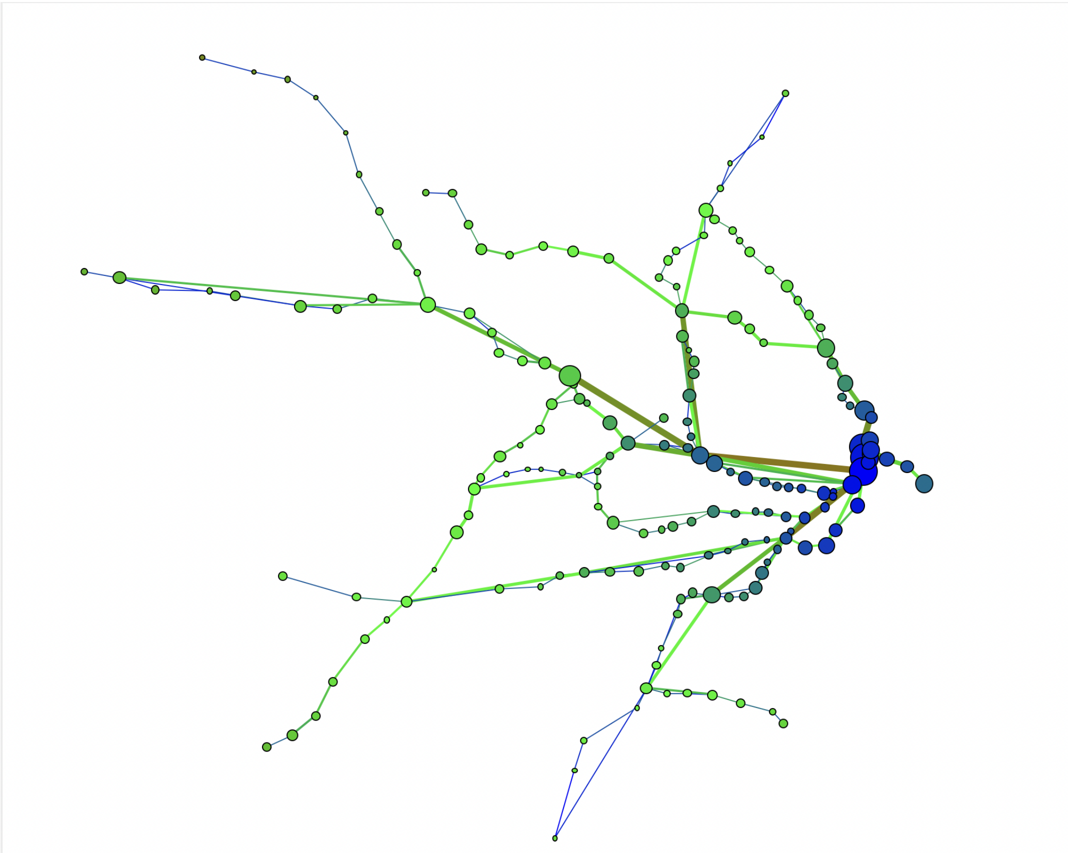


FIGURE 5.2: GUI Demonstration I

The colour of the nodes is based on the time taken to reach from Central Station. The edge thickness and colour is based on expected traffic levels. The size of the nodes is based off total daily traffic at each node.

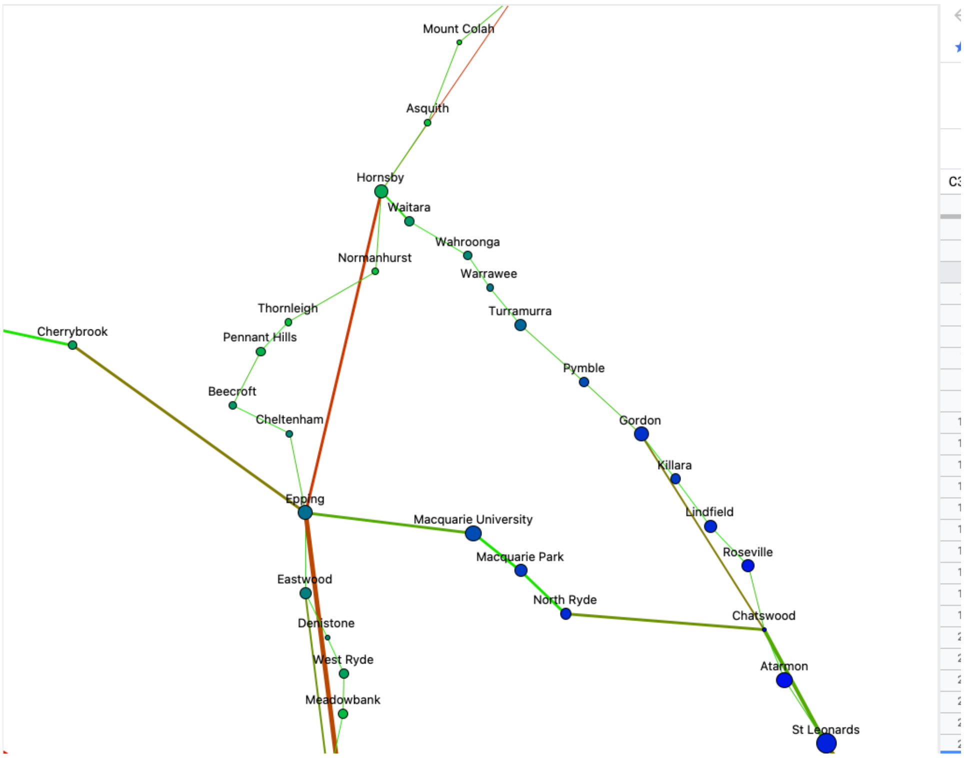


FIGURE 5.3: GUI Demonstration II

Zooming in on the northern regions of Sydney and turning on Display Node Names, we can more easily make out individual stations. The edge thickness is based on expected traffic levels, colour is based on length of the edges. The size of the nodes is based on traffic travelling to Chatswood from each Node, while the colour of the node is based on the distance of each Node from Chatswood.

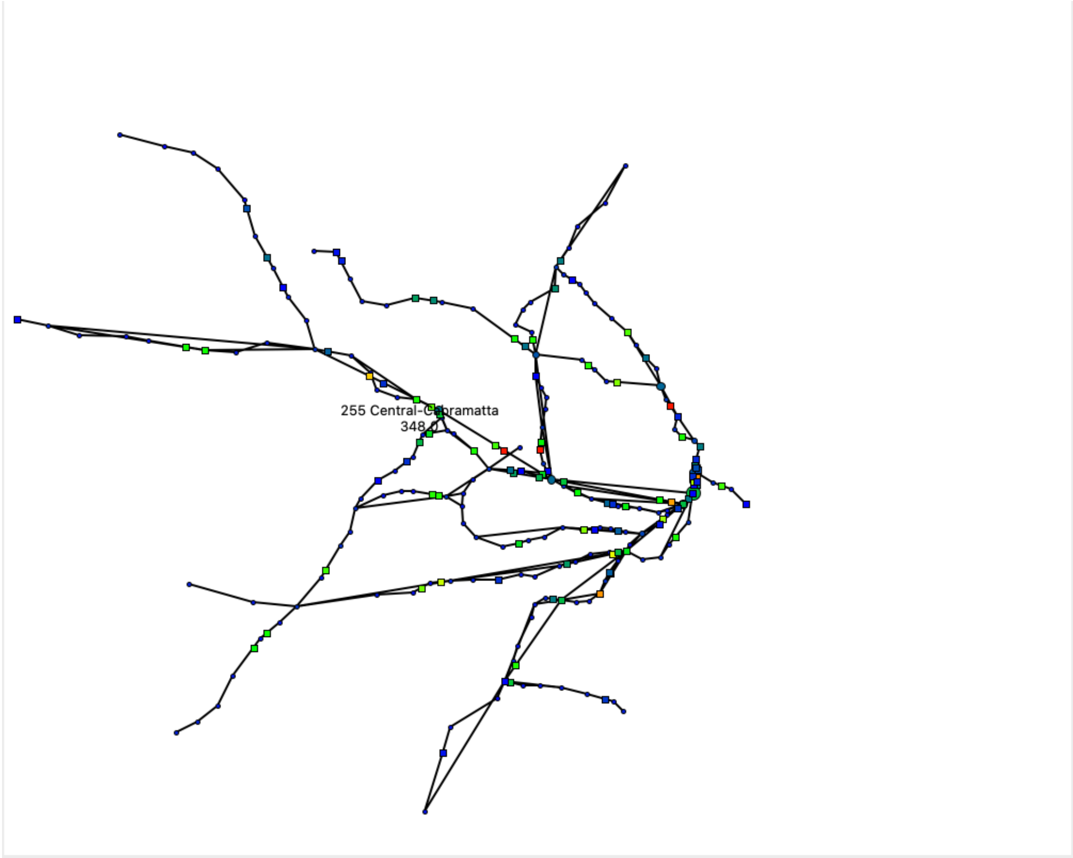


FIGURE 5.4: GUI Demonstration III

After running the simulation, we can observe the simulation results. The diamond shape vehicle have colours that vary with crowding.

CHAPTER 6

Evaluation of Framework on Sydney Trains Network

6.1 Overview of Sydney Trains Network

The Sydney Trains system is a public transport which serves Sydney, the capital of New South Wales and most populous city in Australia. In 2018-2019 it had patronage of 377.1 million passenger journeys [NSW 2019a]. It plays a key role in transporting Sydneysiders around responsible for 16.2% of trips to work in the Greater Sydney Region [Decisions 2016], and a much higher proportion of trips into the Central Business District.

It is very large and complex network. It features 179 individual stations, connected with 935 km of electrified track [NSW 2019a]. The network consists of numerous branch's, featuring both hub and spoke and cross-suburban connections. Of the lines in the network (which are more of an administrative classification than one reflecting actual operation), all but one are served by manually driven double-decker electric trains. The outlier is the Sydney Metro from Chatswood to Tallawong, which is served by automated single decker electric trains.

For the purpose of evaluating our framework, we will construct a model of the Sydney Trains network. We will maintain the full complexity of the routes of the network, however to avoid additional complexity in the optimiser we will assume that all lines including the Sydney Metro are served by manual double decker trains with identical parameters.

For our "control" manually built timetable, we will be using a modified version of the real Sydney Trains Timetable as of August 2022. These modifications will retain the full variety of routes in the original, however variation of frequency will be removed to make it more comparable to the optimiser generated timetable, as our evaluated optimiser does not support changing frequency of services throughout the day.

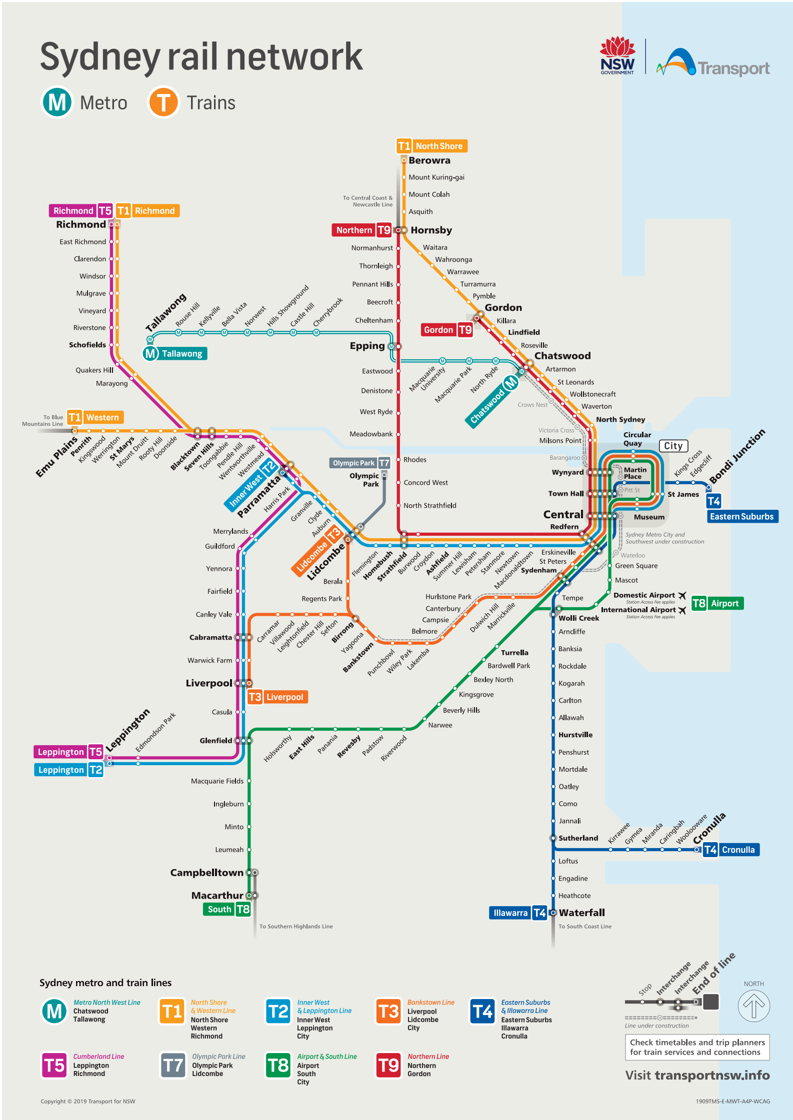


FIGURE 6.1: Map of Sydney Trains System, [NSW 2019b]

6.2 Scenario Description

Using this model, we will compare both a simplified version of the real Sydney Trains schedule (Manual Timetable) and timetables designed by the previously described Henry Convex Optimiser. This was be done on two separate scenarios. In both scenarios we will model the system for 21 hours, reflecting the real Sydney Trains typical operation from 4am to 1am the next day. No new vehicles will be assigned after midnight, however existing vehicles will be allowed to finish their routes.

In the first scenario, the amount of passenger traffic will be constant throughout the day, apart from the first and last two hours of operation where passenger generation will linearly increase from zero at the start and linearly decrease back to zero at the end.

In the second scenario, the amount of passenger traffic generated during the day will follower a multiplier for each hour based on the amount of traffic the real Sydney Trains system experienced through the course of an average weekday. This information was extracted from [NSW 2018a]

These results can be easily replicated using the GUI as described in the previous chapter. The experimental scenarios can be chosen by inputting the default **ScenarioFixed.csv** scenario file for the fixed scenario. Modify the scenario entry widget to import **ScenarioWeek.csv** scenario file to replicate the variable scenario.

Once we ran these experiments, we run the evaluator to obtain comparison statistics. We also viewed the results using the GUI to see if there were any interesting patterns not present in the summary statistics.

6.3 Input Data

6.3.1 Node Determination - Station Information

The geographical position of stations as latitude/longitude coordinates was extracted from Google Maps [GOOGLE 2022]. This information was included in the **Nodes.csv** file as described previously.

6.3.2 Edge Determination - Services between Stations

The edges were first extracted from the connections between stations shown in Figure 6.1 [NSW 2019b]. As direct services skipping intermediate stations are modelled as an edge directly connecting the two nodes in our system, we extracted these direct edges by careful evaluation of the Sydney Trains schedule as presented by [Tripview 2022]. The amount of time taken to traverse each edge was also extracted from this schedule. The extracted information was included in the **Edges.csv** file as discussed previously.

6.3.3 Manual Schedule Determination

We extracted both the routes which trains run and the frequency from [Tripview 2022]. As mentioned previously, we did not consider variation of frequency throughout the day so we used the most common frequency along a particular route. We also did not include the unusual services found at the start and end of the day to position vehicles, as vehicle positioning was not something considered in this comparison. The information was included in the **Schedule.csv** and **Schedule_Segments.csv** file as discussed previously.

6.3.4 Passenger Behavior Determination

We extracted the average daily passenger entries for each station from [NSW 2018a]. This information was included in the **Nodes.csv** file as discussed previously. Using the Gravity Model, the subsequent expected traffic between each station pair can be calculated by our framework. The resulting origin-destination passenger statistics are then multiplied by a time varying modifier (which depends on the scenario selected) to obtain how many passengers need to be generated each time-step. The multipliers for each scenario were included in the **ScenarioFixed.csv** and **ScenarioVariable.csv** files as previously discussed. A comparison of the total traffic level generated by each timetable is included in figure 6.2.

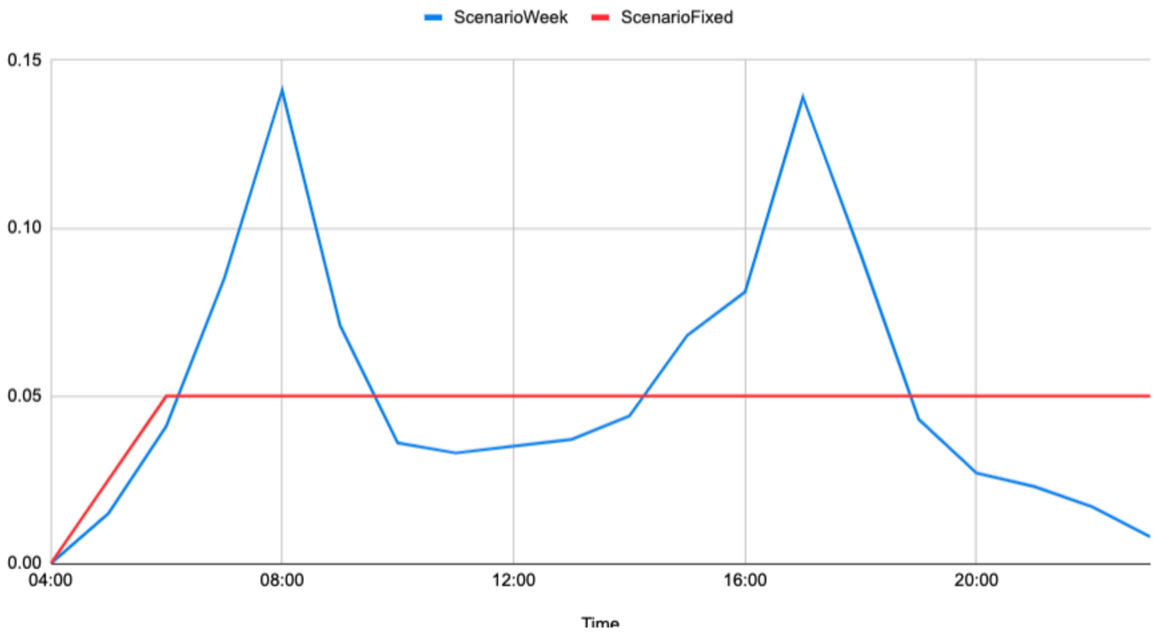


FIGURE 6.2: Variation of traffic between the two scenarios over time

6.3.5 Vehicle Parameters

We used the parameters of the current most common Sydney Trains Vehicle, the 8-carriage Waratah (also called the A/B series) from [NSW 2018b]. We are provided with a seating capacity per train of 910. Using the reasonable assumption of 4 passengers per square meter of space not taken up by seats, we obtain an additional capacity of about 700 standing passengers, for a total capacity of 1,610. Representations of the centre and driving carriages are provided in figures 6.3 and 6.4. A train consists of six centre carriages and two driving carriages, one at each end.

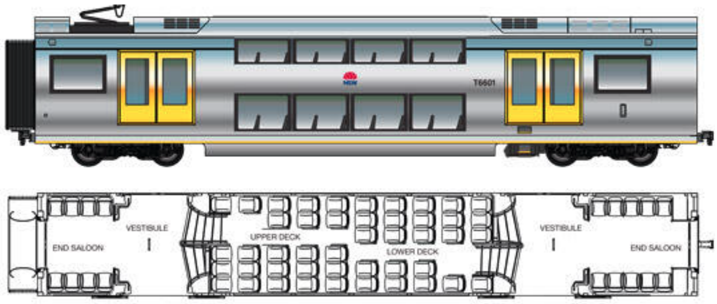


FIGURE 6.3: Diagram of Waratah Train Centre Carriage [NSW 2018b]

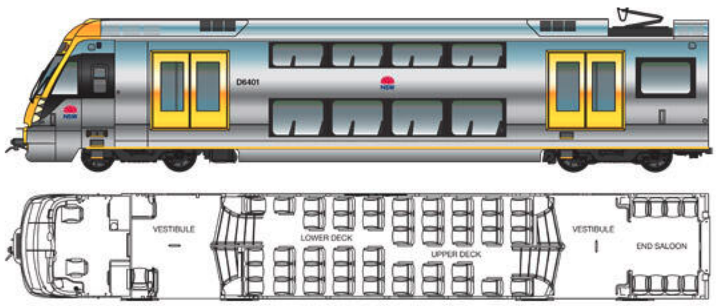


FIGURE 6.4: Diagram of Waratah Train Driving Carriage [NSW 2018b]

6.3.6 Costs Determination

The approximate value of passengers time was found in [Legaspi1 and Douglas 2015] as being between \$10 and \$15 per hour for users of Sydney public transport. We used the lower figure to determine the value of seated passengers time and the upper value for standing passengers and those waiting at stations. This is because being seated is more pleasant than standing, and one can more effectively perform other activities like reading, doing work or wasting time on ones phone. Hence the dis-utility of being seated on a service is less than standing or waiting for a service.

We note from [EDI 2016] that a contract to produce 24 Waratah Trains and maintain them for 35 years was costed at 1.7 Billion Dollars. This gives an all inclusive cost of owning a train for a year at approximately 2.02 million dollars. If we assume that a train operates for 19 hours a day on average (our scenarios run for 21 hours a day, but not all trains are used all the time especially at the start and the end), and assuming that the trains are available 90% of the time (trains are not always available due to maintence, etc), then we obtain an hourly cost of the train itself as $323 per hour.

The train must also be manned in Sydney by a driver and a guard. According to [Commission 2018], typical pay is about $50 per hour. We added an extra 20 % for superannuation and assuming that for every hour operating a train in passenger service roughly half an hour is needed for other tasks such training, starting and planning shifts, breaks and moving trains into and out of storage. This gives us a cost of $90 per hour per crew member for a total labour cost of $180 per hour per train.

We note from [EDI 2021] that the maximum power draw of a Waratah Train is 4’000 KW for traction and 153 KW for non-traction systems. Of course the maximum traction power draw is only used during acceleration, much less power is used during cruise and one while stopped at a station. Assuming that traction power is about one third the stated maximum, we obtain an average power use of about 1’490 kW. Noting the average wholesale power cost has generally been about $100 per MWH in Australia in recent years, we obtain an extra cost due to power of about $149 per hour OPENNEM 2022

Adding these costs together, we obtain a total marginal cost of train operation of about $ 652 per hour.

We included the value of passenger time and the marginal cost of vehicle operation in the **evaluation.csv** file as discussed previously.

All these are of course ballpark figures for the Sydney Trains System, and should be updated with more accurate costs for the specific network when a more detailed evaluation is being performed.

CHAPTER 7

Results and Discussion

7.1 First Scenario - Fixed Traffic

7.1.1 Comparison of Generated Timetable

We first note the difference in the service frequency (gap between two vehicles running the same route) in the simplified Sydney Trains timetable and Timetable generated by the Henry Convex Optimiser. Note the Henry Convex Optimiser uses the mean traffic level. This is shown in table 7.1.

Service Name	Henry Convex	Original Timetable
Berowra-Central	12	30
Hornsby-Central (Chatswood)	10	30
Gordon-Central	9	15
Hornsby-Central (Epping)	14	15
Epping-Central	14	15
Tallawong-Chatswood	11	10
Bondi Junction-Central	5	10
Emu Plains-Central	15	15
Richmond-Central	22	30
Schofields-Central	18	30
Blacktown-Central	13	15
Parramatta-Central	12	15
Flemington-Central	10	15
Macarthur-Central	13	15
Revesby-Central	11	15
Cronulla-Central	16	15
Waterfall-Central	18	30
Sutherland-Central	16	30
Mortdale-Central	13	15
Sefton-Central	17	30
Bankstown-Central	16	30
Campsie-Central	14	15
Leppington-Parramatta	17	30
Liverpool-Parramatta	15	30
Liverpool-Central	19	15
Cabramatta-Central	18	15
North Sydney-Central	6	5
City Circle	6	5
Lidcombe-Oylimpic Park	14	15

TABLE 7.1: Comparison of Service Frequency between Fixed Timetable and Henry Convex Optimised Timetable in the Fixed Traffic Scenario

As we can see in Table 5.1, the convex optimisation approach suggests more or similarly frequent services than the real Sydney Trains timetable in most cases. The main exceptions, namely North Sydney-Central and the City Circle, are very high patronage lines with frequent services in both cases.

Based on this higher frequency of services, we would expect that the Henry Convex Timetable would have shorter passenger wait times and hence the "cost" of passenger time would be lower. However the higher frequency should also increase the number of vehicles required and hence the financial cost of running the system. The overall combined cost should be lower in the optimised case. Running the evaluation engine on both simulations, we can see how accurate our predictions are. The results of this can be seen in Table 7.2

7.1.2 Results

Service Name	Original Timetable	Henry Convex
Num Passenger Trips	1,286,470	1,287,071
% Trips Failed	1.59	1.64 %
Total Time per Passenger (mins)	34.16	32.11
Time Standing (mins)	0.16	0.12
Time Seated (mins)	24.62	24.79
Time Waiting (mins)	9.38	7.20
Cost of Vehicle Operation $	1,739,112	2,130,388
Max Vehicles at Once $162		181
Max Passengers in a Vehicle	1,423	1,098
Combined Vehicle + Time Cost $	7,903,163	7,682,183
Vehicle Cost per Head $	1.35	1.66
Combined Cost Per Head $	6.14	5.97
Simulation Time (seconds)	213.3	243.7

TABLE 7.2: Comparison between Fixed Timetable and Henry Convex Optimised Timetable in the Fixed Traffic Scenario

7.1.3 Discussion

From the results in Table 5.2 we can see that the results lined up with our expectations. The optimiser produced slightly shorter trips for passengers due to shorter waiting times. The higher frequency of vehicles increased the financial cost of the system, however the combination of the cost of passenger time and financial cost of running the network was lower in the optimised scenario.

The Optimisers focus on increasing frequency in busier parts of the network can be seen with the recorded maximum number of passengers in a vehicle, which is much lower in the Henry Convex scenario than the Original Timetable.

It should be noted that as we are dealing with a simplified model, these results should not be taken as evidence that the current Sydney Trains timetable is sub-optimal. Instead this should be scene simply as a demonstration of our framework for an optimisation problem.

7.2 Second Scenario - Variable Traffic

7.2.1 Comparison of Generated Timetable

We once again note the difference in the service frequency between the simplified Sydney Trains Timetable and the Henry Convex Optimised Timetable. The convex optimisation was done using the average traffic level. This comparison can be seen in table 5.3

Service Name	Henry Convex	Original Timetable
Berowra-Central	12	30
Hornsby-Central (Chatswood)	11	30
Gordon-Central	10	15
Hornsby-Central (Epping)	15	15
Epping-Central	15	15
Tallawong-Chatswood	11	10
Bondi Junction-Central	5	10
Emu Plains-Central	16	15
Richmond-Central	23	30
Schofields-Central	18	30
Blacktown-Central	14	15
Parramatta-Central	13	15
Flemington-Central	11	15
Macarthr-Central	13	15
Revesby-Central	11	15
Cronulla-Central	17	15
Waterfall-Central	19	30
Sutherland-Central	17	30
Mortdale-Central	13	15
Sefton-Central	18	30
Bankstown-Central	17	30
Campsie-Central	15	15
Leppington-Parramatta	17	30
Liverpool-Parramatta	15	30
Liverpool-Central	19	15
Cabramatta-Central	18	15
North Sydney-Central	6	5
City Circle	6	5
Lidcombe-Oylimpic Park	14	15

TABLE 7.3: Comparison of Service Frequency between Fixed Timetable and Henry Convex Optimised Timetable in the Variable Traffic Scenario

TWe can see that the optimised timetable has changed very little. This is unsurprising as mean traffic levels are very similar across the two scenarios, even as the distribution is different. However as can be seen in figure 6.2, the distribution of traffic is very different.

7.2.2 Results

Service Name	Original Timetable	Henry Convex
Num Passenger Trips	1,405,365	1,407,562
% Trips Failed	0.18	0.18 %
Total Time per Passenger (mins)	39.45	31.82
Time Standing (mins)	0.26	0.12
Time Seated (mins)	23.41	24.57
Time Waiting (mins)	15.78	7.13
Cost of Vehicle Operation $	1,739,112	2,234,241
Max Vehicles at Once $162		187
Max Passengers in a Vehicle	1,610	1,610
Combined Vehicle + Time Cost $	7,836,440	5,192,111
Vehicle Cost per Head $	1.24	1.59
Combined Cost Per Head $	5.57	3.69
Simulation Time (seconds)	249.6	254.6

TABLE 7.4: Comparison between Fixed Timetable and Henry Convex Optimised Timetable in the Variable Traffic Scenario

7.2.3 Discussion

Regardless of the optimiser used, the variable scenario is noticeably different from the fixed scenario. The number of failed passenger is significantly reduced as passengers are generally only unable to complete their journeys if they start their journey near the end of the simulation. As the variable scenario, in keeping with the real world data, has a much lower number of passengers generated late at night, this is less of a problem in the variable scenario. This in turn produces a much total cost as the penalties applied for a failed journey a steep.

Also notable is that the simulated peak periods result in the network being overcrowded for a brief period. This can be seen as in both tests the maximum number of passengers in a vehicle is at the maximum , indicating some passengers were unable to board the first vehicle to their destination.The higher frequencies in the optimised scenario meant the the number of passengers affected were very small and waiting and standing times were not really affected compared to the fixed scenario where there was no overcrowding. However in the fixed timetable scenario, dramatic overcrowding was experienced in the peaks, almost doubling the waiting time as passengers are unable to board the first service. This is an unsurprising result as we are throwing weekday traffic at a timetable designed for a weekday, something it was not designed to handle. This overcrowding can be seen in figure 7.1

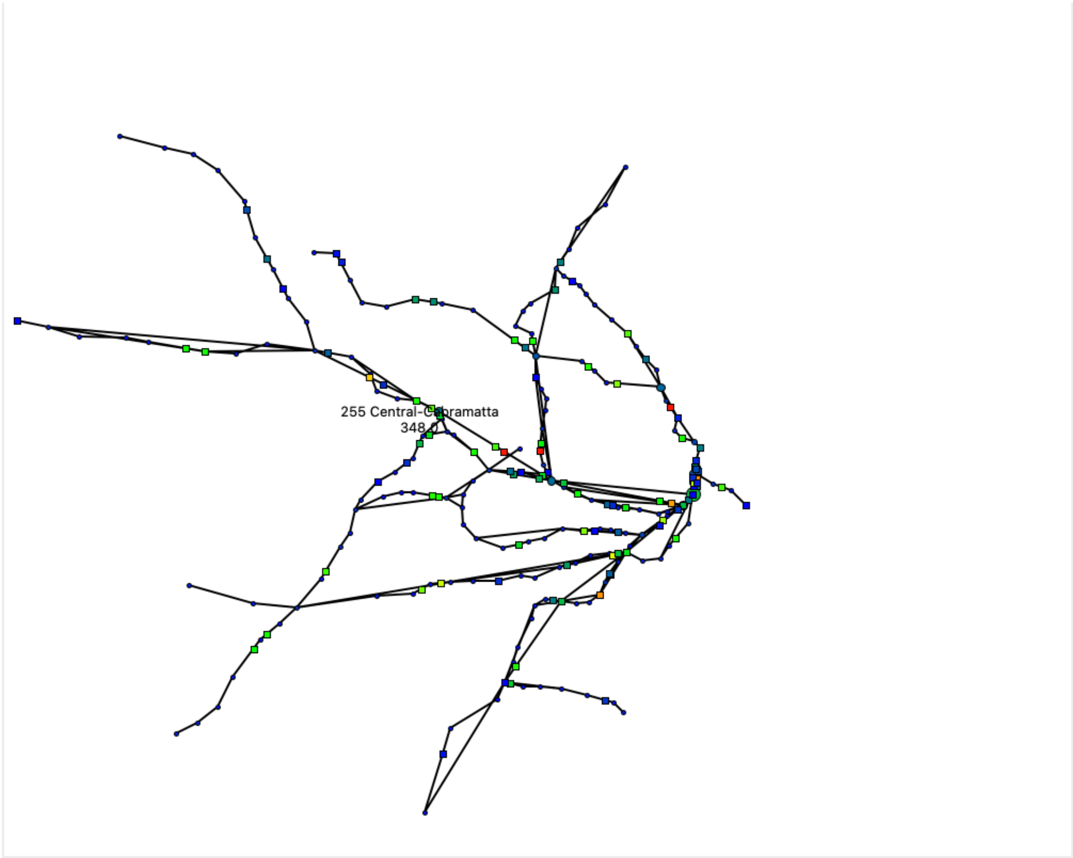


FIGURE 7.1: Network Crowding at 9am in the Fixed Timetable Variable Traffic Scenario

The red and orange colour of many of the vehicles indicates overcrowding.

This result clearly shows the impact that very high traffic levels can have on a network, and hence the importance of varying frequency of service with passenger volume, as outside of the peaks, vehicles were mostly empty while inside them, they were very crowded.

7.3 Strengths and Weaknesses of Framework

We have developed a robust software framework for evaluating public transport optimisers using the model described earlier. It demonstrates good computational performance, taking approximately three minutes to perform the simulations mentioned in the previous section (Using a 2020 M1 Macbook Air). Given that this simulation requires simulating over a million passengers who must find a path across a complex network with hundreds of nodes and edges, this is good performance for most applications. However faster performance would be very useful as it would make optimisers that need to use simulation results (eg genetic algorithms) much more viable.

We have also built an extensive graphical user interface to enable the user to control the software, configure the simulation and evaluate the results. This enables us to see in real time how passengers and vehicles move through the network, allowing end users to understand how traffic moves through a network and visually see how a proposed schedule or optimiser performs. The ability to zoom in makes it easier to focus in on particular areas of the network, which is very helpful on larger networks where the amount of information on the screen when looking at the full network can be overwhelming. As mousing over a node, edge or vehicle provides some information about it including it's name, this also allows a user to easily tell what part of the network they are looking at.

However there are at present no tools to manually modify schedules in the GUI, if the user wishes to modify these they will need to modify the CSV files. An additional interface to modify schedules and even routes would be very helpful to less technically included end users. Additionally there are no tools in the GUI to automatically highlight where "things are going wrong" in terms of overcrowded trains/stations, this can be a problem for larger and longer running simulations are it is easy to miss such problems through manual inspection.

In regards to the simulation itself, there are currently no way of implementing physical constraints in terms of setting a minimum headway (time between vehicles) along a particular route. This trusts that the user or optimiser builds a timetable which does not have place vehicles dangerously close to each other. This is not a problem for the evaluations done so far, however it would become more important when dealing with higher throughput networks or when disruptions to the network are simulated. Implementing physical constraints would also allow for optimisers that design their own routes to be implemented.

At the moment, overall passenger travel levels can be varied throughout the day to simulate peak and off peak period. However in real systems not just the volume of traffic changes, but also the frequency of origins and destinations. Eg commuters travel into work from the suburbs in the morning and home in the evening. Hence allowing more fine-grained control of when and where traffic varies would be very beneficial.

The simulation also assumes that passengers are loaded and unloaded from vehicles instantly, while in reality it takes much longer to unload a crowded train than an empty one. The simulation is also not setup to handle random disruption as could be caused either by overcrowding or by random events (eg mechanical failure). As we wish to ensure that public transport systems are not just optimal but also robust, this is a key failing.

The simulation also assumes passengers are able to change vehicles at intersections instantly. While a reasonable assumption when dealing with smaller stations, at large complex stations, eg Central in Sydney, it could take many minutes to change platforms.

The simulation also assumes that all vehicles in a network have identical parameters (through the parameters of all vehicles can be varied). While this is a good enough assumption for even complex urban rail systems where all vehicles are generally very similar, it would be a much more flawed when dealing with nationwide rail systems or air travel, where differences in type of vehicle are much more significant.

While the evaluation function considers the dis-utility of standing vs sitting vs waiting for a service, this is not considered by the simulation and path-finding, with passengers only focusing on finding the shortest route. This stops from the simulation from modelling that some passengers may prefer to wait and board a less crowded service, or use a more comfortable form of transport.

7.4 Future Improvements

While our software framework provides considerable ability to model complex public transport systems, there are many potential enhancements which could improve it further, enabling more accurate simulation and evaluation of a public transport system. To facilitate this, the software is designed to be easily extensible in the future, aided by the use of modular object orientated programming. Potential future improvements include.

- Making vehicles take time to load/unload passengers, allowing for more accurate simulation of the flow on effects of an overcrowded network. This would also require decreasing the length of the time-step from one minute.
- Allowing for not just the total volume of traffic but the volume of traffic too or from specific nodes to vary throughout the day.

- Making passenger utility vary not just with time taken to reach the destination, but also the level of crowding they experience on their trip.
- Ability to add in random disruptions to schedules during simulation (both major and minor) to better determine how resilient the system is too disruption.
- Implement more complex optimisation algorithms, to see how well they perform using our more complex simulation methods.
- Upgrade path-finding algorithms (which at present are the most computationally intensive part of the program) to improve performance. This could include implementing advanced techniques like contraction hierarchies or rewriting core parts of the program in lower level languages C/C++.
- Implement some of the optimisation techniques mentioned in the literature review using the framework, eg Tabu Search, SSN, Genetic Algorithms, Simulated Annealing.
- Upgrade the GUI to allow manual modification of schedules and routes in the GUI
- Add tools to the GUI to make it very easy to see where things are "going wrong", eg overcrowded trains and stations.
- Make it take time to switch between vehicles at intersections, this time requirement could even vary based on how crowded the intersection is.
- Add in physical constraints to how vehicles can move around the network, which would helpful to better model disruptions and allow for optimisers that design their own routes
- Make it possible for the simulation to simulate different types of vehicles in the same network.
- Make the path-finding consider the dis-utility of different levels of crowding/different vehicles, allowing for a simulation of some passengers preferring to wait to catch a less crowded service.

7.5 Conclusion

In conclusion, our new framework offers a great improvement in the accessibility of software to test and evaluate public transport schedule optimisers. It already is capable of modeling fairly complex systems as demonstrated in the results section. However considerable additional effort is needed to allow it to utilise more complex optimisation techniques and better simulate the complexity of a real public transport network.

Bibliography

Ahmed, Bayes (2012). 'The Traditional Four Steps Transportation Modeling Using Simplified Transport Network: A Case Study of Dhaka City, Bangladesh'. In: *IJASETR* 1 (1). URL: https://escholarship.org/content/qt7j0003j0/qt7j0003j0.pdf.

Bonabeau, Eric (2002). 'Agent-based modeling: Methods and techniques for simulating human systems'. In: *PNAS* 99 (3), pp. 7280–7287. URL: https://www.pnas.org/doi/full/10.1073/pnas.082080899.

Commision, NSW Fair Work (2018). 'Sydney Trains Enterprise Agreement 2018'. In: URL: https://locoexpress.com.au/wp-content/uploads/2018/05/Sydney-Trains-decision-EA%.pdf.

Decisions, Informed (2016). *City of Sydney Method of Travel to Work*. URL: https://profile.id.com.au/sydney/travel-to-work.

Developers, Numpy (2022). *Numpy*. URL: https://numpy.org/doc/stable/index.html.

Dibbelt, Julian, Ben Strasser and Dorothea Wagner (2014). 'Customizable Contraction Hierarchies'. In: DOI: 10.48550/ARXIV.1402.0402. URL: https://arxiv.org/abs/1402.0402.

Dijkstra, Edsger W (1959). 'A note on two problems in connexion with graphs'. In: *Numerische mathematik* 1.1, pp. 269–271.

Doran, J. E. and D. Michie (Sept. 1966). 'Experiments with the Graph Traverser Program'. In: *Proceedings of the Royal Society of London Series A* 294.1437, pp. 235–259. DOI: 10.1098/rspa.1966.0205.

EDI, Downer (2016). *Downer Awared Sydney Growth Trains Contract*. URL: https://web.archive.org/web/20170220042916/http://www.downergroup.com/Investors-and-media/ASX-announcements/2016/Downer-awarded-Sydney-Growth-Trains-contract.aspx.

— (2021). 'Sydney Growth Trains: Enviromental Product Declaration'. In: URL: https://epd-australasia.com/wp-content/uploads/2018/09/24130-EPD-for-Sydney-Growth-Trains-FA-v2_11Oct18.pdf.

Fan, Wei and Randy B. Machemehl (2006). 'Using a Simulated Annealing Algorithm to Solve the Transit Route Network Design Problem'. In: *Journal of Transportation Engineering* 132.2, pp. 122–132.

Foundation, Python Software (2022). *Python3*. URL: https://www.python.org/about/.

Fredman, M.L. and R.E. Tarjan (1984). 'Fibonacci Heaps And Their Uses In Improved Network Optimization Algorithms'. In: *25th Annual Symposium onFoundations of Computer Science, 1984*. Pp. 338–346. DOI: 10.1109/SFCS.1984.715934.

Gamma, Erich et al. (1994). *Design Patterns: Elements of Reusable Object-Orientated Software*. Addison-Wesley.

Geisberger, Robert et al. (2008). 'Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks'. In: *Experimental Algorithms*. Ed. by Catherine C. McGeoch. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 319–333.

Glover, Fred W (1986). 'Future Paths for Integer Programming and Links to artificial Intelligence design.' In: *Computers and Operations Research.* 13 (5), pp. 533–549.

GOOGLE (2022). *Google Maps*. URL: https://www.google.com/maps.

J, Bookbinder and Desilets A (2002). 'Transfer Optimization in a Transit Network'. In: *Transport Science* 26 (2), pp. 106–118.

Kagho, Grace O., Milos Balac and Kay W. Axhausen (2020). 'Agent-Based Models in Transport Planning: Current State, Issues, and Expectations'. In: *Procedia Computer Science* 170, pp. 726–732. URL: https://www.sciencedirect.com/science/article/pii/S187705092030627X.

Legaspi1, Julieta and Neil Douglas (2015). 'Value of Travel Time Revisited - NSW Experiment'. In: URL: https://www.transport.nsw.gov.au/sites/default/files/media/documents/2017/Value%5C%20of%5C%20Travel%5C%20Time%5C%20Revisited%5C%20%5C%E2%5C%80%5C%93%5C%20NSW%5C%20Experiment.pdf.

Manheim, Marvin L. (1979). *Fundamentals of Transportation Systems Analysis*. MIT Press.

Marks, Dr Benjy (2020). *University of Sydney Thesis Template (Engineering)*. URL: https://www.overleaf.com/latex/templates/university-of-sydney-thesis-template-engineering/hchhbxdtdctm.

McNally, Micheal G. (2000). 'Four Step Model'. In: *Institute of Transportation Studies*. URL: https://escholarship.org/content/qt7j0003j0/qt7j0003j0.pdf.

NSW, Transport for (2018a). *Opal Tap on and Tap Off*. URL: https://opendata.transport.nsw.gov.au/dataset/train-station-entries-and-exits-data.

— (2018b). *Waratah Trains*. URL: https://transportnsw.info/travel-info/ways-to-get-around/train/fleet-facilities/waratah-trains.

— (2019a). *Sydney Trains Annual Report 2018-2019*. URL: https://www.transport.nsw.gov.au/news-and-events/reports-and-publications/sydney-trains-annual-reports.

— (2019b). *Sydney Trains Network Map*.

NumFocus (2022). *PANDAS*. URL: https://pandas.pydata.org/.

OPENNEM (2022). 'Open National Electricity Market'. In: URL: https://opennem.org.au.

P, Chakroborty (2003). 'Genetic algorithms for optimal urban transit network design'. In: *Computer Aided Civil Infrastructure Engineering* 18 (3), pp. 184–200.

Parry, Hazel R. (2009). 'Agent Based Modeling, Large Scale Simulations'. In: *Encyclopedia of Complexity and Systems Science*. Ed. by Robert A. Meyers. New York, NY: Springer New York, pp. 148–160. ISBN: 978-0-387-30440-3. DOI: 10.1007/978-0-387-30440-3_9. URL: https://doi.org/10.1007/978-0-387-30440-3_9.

Pincus, Martin (1970). 'A Monte Carlo Method for the Approximate Solution of Certain Types of Constrained Optimization Problems'. In: *Journals of the Operations Research Society of America* 18 (6), pp. 967–1235.

Rodriguez, D., Ana Olivera and Nélida Brignole (July 2014). 'VEHICLE ROUTING FOR PUBLIC TRANSPORT WITH ADAPTED SIMULATED ANNEALING'. In: *Latin American applied research Pesquisa aplicada latino americana = Investigación aplicada latinoamericana* 44, pp. 247–252.

S, Voss (1992). 'Network design formulation in schedule synchronization'. In: *Computer-aided scheduling of public transport Springer*, pp. 137–152.

Șerban, Alin (Feb. 2021). 'The use of the genetic algorithms for optimizing public transport schedules in congested urban areas'. In: 1037.1, p. 012062. DOI: 10.1088/1757-899x/1037/1/012062. URL: https://doi.org/10.1088/1757-899x/1037/1/012062.

Strandberg, Jani, Saku Metsarinne and Sami Kari-Koskinen (2016). 'Pathfinding in agent-based people flow simulation'. In: *Alto University School of Science*. URL: https://sal.aalto.fi/files/teaching/ms-e2177/2016/KONEfinal.pdf.

Tripview (2022). *Tripview*. URL: http://www.tripview.com.au/.

APPENDIX A

Appendix

A1 Appendix A: Code Listings

Code developed in this thesis is included below. This code is also available online at https://github.com/henryc47/Thesis_Public_Transport_Optimisation

A1.1 main.py

```
#main.py
#this runs the simulation

#this code runs a bunch of terminal commands to setup our GUI
def main():
    import render as r
    a = r.Display()
    finish = input('exit?\n')

if __name__ == '__main__':
    main()
```

A1.2 network.py

```
#network.py
#stores information about the physical network

import warnings #for warnings
```

```python
import numpy as np #for large scale mathematical operations
import time as time #for benchmarking
import schedule as schedule
import vehicle as vehicle
import copy as copy #for shallow-copying schedules
import random as rand
rand.seed(30699) #consistent seed to ensure consistent results
import agent as a

#edge class, represents a (one-way) link between two nodes
#at the moment, only relevant property is travel time taken, but more properties may be
    added later
#we will be using one second increments for time
class Edge:
    #initialise the node
    def __init__(self,name,start_node,end_node,travel_time):
        self.name = name
        self.start_node = start_node
        self.end_node = end_node
        self.travel_time = travel_time

    #provide the destination of the link
    def provide_destination(self):
        return self.end_node

    #provide the time to travel along the link
    def provide_travel_time(self):
        return self.travel_time

    #provide information about the edge, for testing purposes
    def test_edge(self):
        print(self.name,' is from ',self.start_node,' to ',self.end_node,' a trip taking
            ',self.travel_time)

#node class, represents a location between which passengers can travel
#the node stores the names of all the nodes which start at it
class Node:
```

```
    def __init__(self,name,coordinates,id,network):
        self.name = name
        self.edge_names = []#list of all edges starting at this node
        self.edge_destinations = []#and the destination of each node
        self.edge_times = []#matching list of travel time of each respective edge
        (self.latitude,self.longitude) = extract_coordinates(coordinates)
        self.agents = [] #list of all agents at this stations
        self.schedule_names = [] #list of schedules stopping at this station
        self.schedule_times = [] #times at which vehicles arrive at this node
        self.nodes_after = [] #list of nodes after this node on a schedule
        self.node_times_after = [] #time to reach nodes after the node on the schedule
        self.id = id #id of the node
        self.network = network #network we belong too
        #has the next vehicle of each schedule arriving at the node changed since we
            lasted found paths
        self.next_vehicle_changed = True #starts at true so that we can use the reset
            variables process to initialise our variables
        self.num_agents = 0

    #add an edge which starts at the node
    def add_edge(self,edge):
        if edge.start_node == self.name:#the edge will be stored with this node only if
            it starts at the node
            self.edge_names.append(edge.name)
            self.edge_destinations.append(edge.end_node)
            self.edge_times.append(edge.travel_time)
            return True#Return true to indicate edge has been associated with the node
        else:
            print('edge ', edge.name, ' between ', edge.start_node, ' and ', edge.
                end_node, ' does not start at node ', self.name)
            return False #Return false to indicate edge has not been associated with the
                node

    #return the time taken to travel along a particular edge
    #for this function to work correctly, edge names must be unique
    def provide_edge_time(self,edge_name):
        try:
```

```
            edge_index = self.edge_names.index(edge_name)
            time_taken = self.edge_times[edge_index]
            return (True,time_taken) #True to indicate search operation was successful
        except ValueError: #edge name not in list of provided eges
            print('edge ', edge_name, ' not in list of edges starting at this node')
            return False #False to indicate search operation unsuccessful

    #return the time taken to travel to all neighbouring nodes and the names of the destination
    def provide_nodes_time(self):
        return (self.edge_times,self.edge_destinations)

    #as above, but also provides the name of the connecting edge
    def provide_nodes_time_edge_name(self):
        return (self.edge_times,self.edge_destinations,self.edge_names)

    #return the time taken to travel to a destination as well as the edge to reach it
    #for this function to work correctly, edge names must be unique
    def provide_node_time(self,destination_name):
        try:
            node_index = self.edge_destinations.index(destination_name)
            time_taken = self.edge_times[node_index]
            edge_taken = self.edge_names[node_index]
            return (True,time_taken,edge_taken) #True to indicate search operation was successful
        except ValueError: #destination name not in list of provided nodes
            print('node ', destination_name, ' not in list of nodes reachable from this node')
            return False #False to indicate search operation unsuccessful

    #add a agent to the station
    def add_agent(self,agent):
        self.agents.append(agent)
        self.num_agents = self.num_agents + agent.number_passengers #the number of passengers has increased

    #remove agent from the station
```

```
    def remove_agent(self,id):
        removed_agent = self.agents.pop(id)
        self.num_agents = self.num_agents - removed_agent.number_passengers #the number of passengers has decreased
        return removed_agent


    #count the number of agents at the station
    def count_agents(self):
        #num_agents = 0
        #for agent in self.agents:
        #    num_agents = num_agents + agent.number_passengers
        return self.num_agents


    #add a schedule which stops at that station
    def add_stopping_schedule(self,schedule_name,schedule_times,node_offset,nodes_after,node_times_after):
        self.schedule_names.append(schedule_name)
        schedule_times_mod = [schedule_time+node_offset for schedule_time in schedule_times] #offset schedule times by time to reach the node
        self.schedule_times.append(schedule_times_mod)
        self.nodes_after.append(nodes_after)
        self.node_times_after.append(node_times_after)

    #calculate the time till the next service of each schedule arrives at a node
    def time_till_next_vehicles(self,current_time):
        num_schedules = len(self.schedule_names)
        next_service_times = []
        for i in range(num_schedules): #go through all the schedules at a node
            #calculate service data for each particular schedule
            schedule_times = self.schedule_times[i]
            num_future_services = len(schedule_times)
            j = 0 #which service are we looking at
            next_service_time = np.inf #default next service time is infinity
            while j<num_future_services:
                service_time = schedule_times[j]
```

```
                if service_time>=current_time:
                    next_service_time = service_time
                    break #we have found a service after (or equal) to the present time, so need to search further
                j = j + 1 #look at the next service
            next_service_times.append(next_service_time)

        return next_service_times

    #remove vehicles which have already arrived at the node
    def remove_arrived_vehicles(self,current_time):
        num_schedules = len(self.schedule_names)
        for i in range(num_schedules): #go through all the schedules at a node
            while len(self.schedule_times[i])>0:
                if self.schedule_times[i][0]<=current_time: #if this service is in the past
                    self.schedule_times[i].pop(0) #remove it from the list of services
                else:
                    break #as services of a schedule are in order, we only need to evaluate till we find a service in the future

    #reset the internal info required for pathfinding
    def reset_pathfinding_info(self):
        self.num_nodes_in_network = len(self.network.node_names)
        self.distance_to_nodes = np.zeros(self.num_nodes_in_network) + np.inf #initial distance to reach all other nodes will be infinite
        self.evaluated_nodes = np.zeros(self.num_nodes_in_network)  #when a node is evaluated the value in this matrix is set to infinite, ensuring that node is never evaluated again
        self.evaluated_nodes_tf = np.zeros(self.num_nodes_in_network) #as above, but evaluated nodes are set to 1
        self.distance_to_nodes[self.id] = 0 #initial distance to reach the starting node is 0
        #create an array to store the paths to all the other nodes
        self.path_to_nodes = [[] for _ in range(self.num_nodes_in_network)] #create an empty nested list of the required length to store paths to nodes

```

```
    def check_evaluated_destinations(self, destination_nodes):
        num_evaluated_destinations = np.sum(np.logical_and(self.evaluated_nodes,
            destination_nodes))
        return num_evaluated_destinations

    #find a path from this node to all nodes where num_passengers_to_node is greater
        than 0
    def find_paths(self, num_passengers_to_node, start_time):
        if self.next_vehicle_changed == True:
            self.reset_pathfinding_info() #restart the pathfinding process if the next
                vehicle arriving at this node has changed
            self.next_vehicle_changed = False #compared to the present, next vehicle has
                not changed
        #get info about vehicles arriving at the starting node
        start_next_service_times, start_nodes_after, start_node_times_after,
            start_schedule_names = self.provide_next_services(data_time=start_time, start
            =True)
        destination_nodes = num_passengers_to_node>0 #determine which nodes we need to
            calculate paths too (I.E those where passengers are actually going)
        num_destinations = np.sum(destination_nodes) #number of destinations we are
            trying to reach
        num_evaluated_destinations = self.check_evaluated_destinations(destination_nodes
            ) #get number of destinations already evaluated
        while True: #loop till we meet an exit conditionx
            expected_distance_to_nodes = self.distance_to_nodes + self.evaluated_nodes #
                set the distance to reach an already evaluated node to be infinite so we
                don't choose it as the minimal node
            min_index = np.argmin(expected_distance_to_nodes) #get the index of the node
                with the lowest expected travel time, evaluate this next
            minimum_distance = expected_distance_to_nodes[min_index] #extract the
                minimum distance from the starting node
            if minimum_distance == np.inf:
                break #break out of the loop, we have explored all the network we can
                    reach
            elif num_evaluated_destinations==num_destinations:
                break #break out of the loop, we have already found paths to all the
                    destinations we wish to reach
```

```
        else:
            #otherwise, explore paths from the minimal node
            current_time = minimum_distance +  start_time#time at which we reach the node currently being evaluated
            if min_index==self.id: #if at starting node, use precalcuated data about services
                #use precalculated data from the starting node
                next_service_times = start_next_service_times
                nodes_after = start_nodes_after
                times_after = start_node_times_after
                schedule_names = start_schedule_names
            else: #otherwise, extract data about the evaluation node at the evaluation time
                next_service_times,nodes_after,times_after,schedule_names = self.network.nodes[min_index].provide_next_services(start=False, data_time=current_time)


        #now it's time to calculate the path to other nodes
        num_schedules = len(next_service_times)
        for i in range(num_schedules):
            #extract nodes and times after for this specific route
            next_service_time = next_service_times[i]
            next_service_name = schedule_names[i]
            route_nodes_after = nodes_after[i]
            route_times_after = times_after[i]
            for j,node in enumerate(route_nodes_after):
                node_index = node.id
                distance_to_current_node_old_path = self.distance_to_nodes[node_index] #what is the current shortest path to the node we are looking at
                distance_to_current_node_new_path = minimum_distance + (next_service_time-current_time) + route_times_after[j] #how long to reach next node through evaluation node
                if distance_to_current_node_new_path<distance_to_current_node_old_path: #we have a better path
```

```
                            self.distance_to_nodes[node_index] = distance_to_current_node_new_path
                            route_to_old_node = self.path_to_nodes[min_index] #extract the path to the evaluation node
                            route_to_new_node = copy.copy(route_to_old_node) #path to the next node is path to the evaluation node + new step
                            route_to_new_node.append(next_service_name) #store the next service we need to catch
                            route_to_new_node.append(node.name) #and when we need to get off that service
                            self.path_to_nodes[node_index] = route_to_new_node #store this in the list of all paths

            self.evaluated_nodes[min_index] = np.inf #mark the node as evaluated, it will not be evaluated again
            self.evaluated_nodes_tf[min_index] = True #as above
            if destination_nodes[min_index]==True:
                num_evaluated_destinations = num_evaluated_destinations+1

        #once we have found the paths to all nodes, return the paths and number of passengers
        #note we return the number of passengers going to an unreachable station as zero, but we return the number of passengers who failed to reach their destination as well
        num_nodes = len(self.network.nodes)
        num_unreachable_passengers = 0 #keep track of the number of passengers who fail to reach their destination
        for i in range(num_nodes):
            if self.distance_to_nodes[i]==np.inf: #if the passenger cannot reach this node
                num_unreachable_passengers = num_unreachable_passengers + num_passengers_to_node[i] #add them to the total of failed passengers
                num_passengers_to_node[i] = 0 #do not create any passengers trying to reach this node

        return self.path_to_nodes, num_passengers_to_node, num_unreachable_passengers
```

```python


    #as previous function, but store the result in a internal variable
    #this is useful for operations at the current time
    def self_time_till_next_vehicles(self,current_time):
        self.next_service_times = self.time_till_next_vehicles(current_time)

    #provide the next service
    def provide_next_services(self,data_time=0,start=False):
        if start==True:
            #we are providing service info at the same time as we are creating a
                passenger, so use precalculated times
            next_service_times = self.next_service_times
        else:
            #otherwise calculate the time dynamically
            next_service_times = self.time_till_next_vehicles(data_time)
        #in either case, we must return the corresponding following nodes and their time
            to reach
        return next_service_times,self.nodes_after,self.node_times_after,self.
            schedule_names

    def test_node(self):
        print('from node ',self.name, ' edges are')
        for i in range(len(self.edge_names)):
            print(self.edge_names[i], ' goes too ',self.edge_destinations[i],' taking ',
                self.edge_times[i])
        print('node latitude is ',self.latitude, ' longitude is ',self.longitude)


#network class, represents the overall structure of the transport network
class Network:
    #initalise the physical network
    #note, this assumes that passengers are evenly distributed through the day
    def __init__(self,nodes_csv,edges_csv,schedule_csv,parameters_csv,eval_csv,
        scenario_csv,verbose=1,segment_csv='',schedule_type='simple',optimiser='
        hardcoded'):
        time1 = time.time()
```

```
        print('optimiser ',optimiser)
        self.verbose = verbose #import verbosity
        #where we will store edges and nodes
        self.edges = [] #list of edges
        self.nodes = [] #list of nodes
        self.edge_names = [] #list of generated edge names
        self.optimiser = optimiser #optimisers we can use, options are "hardcoded", the set frequency from the schedule and "henryconvex", my own custom convex optimisation function
        #extract the raw data
        #now extract node data
        self.node_names = nodes_csv["Name"].to_list()
        node_positions = nodes_csv["Location"].to_list()
        #and let's create the nodes
        num_nodes = len(self.node_names)
        for i in range(num_nodes):
            self.nodes.append(Node(self.node_names[i],node_positions[i],i,self)) #nodes id is it's position in the array

        #extract edge data
        self.edge_starts = edges_csv["Start"].to_list()
        self.edge_ends = edges_csv["End"].to_list()
        self.edge_times = edges_csv["Time"].to_list()
        self.edge_bidirectional = edges_csv["Bidirectional"].to_list()
        #and let's create the edges
        num_edges = len(self.edge_starts)
        for i in range(num_edges):
            if (self.edge_bidirectional[i]=='Yes'):#if input edge is two-way
                #create two edges, one "UP" (by convention towards central), one "DOWN", (away from central)
                self.add_edge(self.edge_starts[i],self.edge_ends[i],self.edge_times[i])#UP
                self.add_edge(self.edge_ends[i],self.edge_starts[i],self.edge_times[i])#DOWN
            else:
                #if input edge is one way
```

```
                self.add_edge(self.edge_starts[i],self.edge_ends[i],self.edge_times[i])#UP

        #extract parameter data
        self.vehicle_max_seated = parameters_csv["Vehicle Max Seated"].to_list()[0] #maximum number who can sit inside a vehicle
        self.vehicle_max_standing = parameters_csv["Vehicle Max Standing"].to_list()[0] #maximum number who can fit inside a vehicle seated + standing
        self.traffic_time_gap = parameters_csv["Traffic Time Gap"].to_list()[0] #gap in timesteps between traffic volume updates
        self.traffic_multiplier = scenario_csv["Traffic Multiplier"].to_list() #traffic multiplier for each time gap of operation
        self.stop_simulation_time = (len(self.traffic_multiplier)-1)*self.traffic_time_gap #time when the simulation should end
        self.vehicle_cost = eval_csv["Vehicle Cost"].to_list()[0] #marginal cost of running a vehicle, $/hour
        self.agent_cost_seated = eval_csv["Agent Cost Seated"].to_list()[0] #marginal value of agents time, $/seated
        self.agent_cost_standing = eval_csv["Agent Cost Standing"].to_list()[0] #marginal value of agents time, higher because standing is unpleasant $/hr
        self.agent_cost_waiting = eval_csv["Agent Cost Waiting"].to_list()[0] #marginal value of agents time, higher because waiting is unpleasant $/hr
        self.unfinished_penalty = eval_csv["Unfinished Penalty"].to_list()[0] #penalty if passengers are unable to reach their destination, based roughly on cost of late night taxi ride
        self.passenger_time_multiplier = float(0) #multiplier on how many passengers are generated per hour, converted to a float as it refuses to become an integer later
        #allocate passengers
        self.node_passengers = (nodes_csv["Daily Passengers"]).to_list()#passengers per day for each station
        time2 = time.time()
        if self.verbose >=1:
            print('time to extract and process network data - ', time2-time1, ' seconds')
        time1 = time.time()
```

```
        self.find_distance_to_all_path()#find the shortest distance between all edges on the network, as well as the paths between them
        time2 = time.time()
        if self.verbose >=1:
            print('time to find ideal travel time between all nodes - ', time2-time1, ' seconds')
        time1 = time.time()
        self.create_origin_destination_matrix()#create the origin destination matrix for the network
        time2 = time.time()
        if self.verbose >=1:
            print('time to assign passengers to origin destination pairs - ', time2-time1, ' seconds')
        time1 = time.time()
        self.find_expected_edge_traffic()
        time2 = time.time()
        if self.verbose >=1:
            print('time to calculate traffic along each edge ',time2-time1, ' seconds')
        #in simple scheduling, schedules are just lists of nodes
        #in complex scheduling, schedules are made up of segments which are lists of nodes
        #note complex schedules are converted to the same immediate format as simple schedules
        self.schedule_csv = schedule_csv
        self.schedule_type = schedule_type #simple schedule type
        self.segment_csv = segment_csv #segments used in the complex schedule
        self.parameters_csv = parameters_csv #segment used
        #setup for vehicle simulations
        self.num_vehicles_started_here = np.zeros(num_nodes) #store the number of vehicles on the network which started from a particular node
        self.vehicles = [] #container to store vehicles in
        self.vehicle_names = [] #container to store vehicle names in, note this is just schedule name followed by initial departure time
        #set the simulation timestamp to be 0 (start of simulation)
        self.time = 0
        #containers to store agents (passengers) and agent ids
        self.agents = []
```

```
        self.agent_ids = []
        self.agent_id_counter = 0 #id of the next agent to be generated
        self.num_failed_agents = 0 #number of agents created who could not find a path and hence were immediately unmade
        self.num_successful_agents = 0 #number of agents who were created and found a path to their destination
        time1 = time.time()
        self.create_schedules() #create the schedules
        if self.optimiser=='hardcoded':
            pass #just use the default schedule gaps from the imported schedule
        elif self.optimiser=='henry_convex':
            self.henry_convex_optimiser() #use this optimiser to generate the schedule gaps
        self.create_dispatch_schedule()
        self.determine_which_nodes_have_schedule() #determine which nodes have which schedules
        time2 = time.time()
        if self.verbose>=1:
            print('time to extract and generate schedules', time2-time1, 'seconds')

    #implemention of my own custom optimisation algorithm
    #which determines the optimal wait time between services based on minimising total service cost + waiting cost
    def henry_convex_optimiser(self):
        schedule_costs = []
        num_nodes = len(self.node_names)
        num_schedules_each_node = np.zeros(num_nodes) #number of schedules at each node
        for schedule in self.schedules:
            length = schedule.get_length()
            cost = (length/60)*self.vehicle_cost #determine the cost of providing a vehicle service
            schedule_costs.append(cost)
            node_names = schedule.node_names #get the name of all the nodes
            for name in node_names:
                node_index = self.get_node_index(name)
                num_schedules_each_node[node_index] = num_schedules_each_node[node_index] + 1 #one more schedule is present at this node
```

```

        #now determine the number of passengers starting at each schedule (nodes with
            multiple schedules have reduced weight)
        for i,schedule in enumerate(self.schedules):
            weighted_passengers = 0
            node_names = schedule.node_names #get the name of all the node
            for name in node_names:
                node_index = self.get_node_index(name)
                node_passengers = self.node_passengers[node_index]*np.mean(self.
                    traffic_multiplier)
                weighted_passengers = weighted_passengers + (node_passengers/
                    num_schedules_each_node[node_index])
            #now use the derived equation (see thesis) to determine the optimal
                frequency
            optimal_wait_time = np.sqrt((2*schedule_costs[i])/(weighted_passengers*self.
                agent_cost_waiting))
            optimal_wait_time = int(optimal_wait_time*60) #convert to integers minutes
            print('for schedule ',schedule.name,' optimal wait time is ',
                optimal_wait_time,' mins') #DEBUG
            self.schedule_gaps[i] = optimal_wait_time



        #having determined the length and weighted number of passengers in each schedule
            , let's calculate the optimal frequency



    #update the passenger time multiplier, sets the number of passengers generated to
        vary throughout the day based on the scenario
    def update_passenger_time_multiplier(self):
        time_period = int(self.time/self.traffic_time_gap)
        time_period_start = time_period*self.traffic_time_gap
        if self.time<self.stop_simulation_time:
            end_time_multiplier = self.traffic_multiplier[time_period+1]
            start_time_multiplier = self.traffic_multiplier[time_period]
        else:
```

```
                end_time_multiplier = 0
                start_time_multiplier = 0

            time_from_start = self.time-time_period_start
            self.passenger_time_multiplier = start_time_multiplier*(1-time_from_start/self.
                traffic_time_gap) + end_time_multiplier*(time_from_start/self.
                traffic_time_gap)
            self.passenger_time_multiplier = self.passenger_time_multiplier

    #create a new vehicle and add it to the network
    def create_vehicle(self,schedule):
        vehicle_name = str(self.time) + " " + schedule.provide_name() #calculate the
            vehicles name
        #produce a shallow copy of the schedule to provide to the vehicle, note we use a
             class defined implemention of shallow-copying
        copy_schedule = copy.copy(schedule) #copy the schedule object, but maintain keep
             references to node/edges identical
        if self.verbose>=1:
            print('schedule destinations ',copy_schedule.nodes)
        junk,start_node = copy_schedule.provide_next_destination() #extract the first
            destination of the schedule
        start_node_index = start_node.id
        self.num_vehicles_started_here[start_node_index] += 1 #record that a vehicle
            started at a particular node
        self.vehicle_names.append(vehicle_name) #add the vehicles name to the list
        self.vehicles.append(vehicle.Vehicle(copy_schedule,self.time,vehicle_name,
            seated_capacity=self.vehicle_max_seated,standing_capacity=self.
            vehicle_max_standing)) #create the vehicle and add it to the list
        if self.verbose>=1:
            print('a vehicle ', vehicle_name, ' has been created at ',start_node.name, '
                 at time ',self.time)

    #this function updates all the vehicle objects in the network
    def move_vehicles(self):
        for count,vehicle in enumerate(self.vehicles):
            #logging
            if self.verbose==1:
```

```
                vehicle.verbose_stop()
            elif self.verbose>=2:
                vehicle.verbose_position()
            not_reached_destination = vehicle.update()
            if not_reached_destination == False:
                if self.verbose>=1:
                    print('a vehicle ', vehicle.name, ' has reached the end of its path
                        at time ', self.time)
                del self.vehicles[count] #remove the vehicle when it has reached it's
                    destination

    #create vehicles at nodes as needed by the schedule
    def assign_vehicles_schedule(self):
        #run through the all the schedules in the dispatch list
        num_schedules = len(self.schedules)
        for i in range(num_schedules):
            if len(self.dispatch_schedule2[i])>0: #if there are still schedules left to
                be dispatched
                if self.time == self.dispatch_schedule2[i][0]: #a vehicle of this
                    schedule is required to be created a the current time
                    self.create_vehicle(self.schedules[i])
                    self.dispatch_schedule2[i].pop(0) #remove the first element of the
                        list as the vehicle has been created at the required time

    #create passengers with pathfinding done at the node level rather than the agent
        level
    def create_all_passengers_pathfinding(self):
        num_nodes = len(self.node_names)
        for i in range(num_nodes): #go through all the nodes we are starting from
            start_node = self.nodes[i] #extract a reference to the starting node
            num_passengers_to_node = np.zeros(num_nodes)
            #calculate the number of passengers going to each node
            for j in range(num_nodes):
                end_node = self.nodes[j] #extract a reference to that node
                num_passengers_pair = self.origin_destination_trips[i,j] #extract number
                    of passengers going between these node pairs
```

```
            num_passengers_per_min = (num_passengers_pair/60)*self.passenger_time_multiplier #we create passengers every minute, but statistics are per hour
            #create the required number of passengers
            int_num_passengers = int(num_passengers_per_min) #rounded-down number of passengers to create
            chance_additional_passenger = num_passengers_per_min-int_num_passengers #chance of an additional passenger being created from the remainder
            num_passengers_to_node[j] = int_num_passengers + random_true(chance_additional_passenger) #get the final number of passengers to be created
        # now determine the path to all the nodes, the number of passengers travelling to each node and the number of passengers which failed to reach their destination
        path_to_nodes, num_passengers_created, num_unreachable_passengers = start_node.find_paths(num_passengers_to_node, self.time)
        self.num_successful_agents = self.num_successful_agents + np.sum(num_passengers_created) #record total successful pathfinding agents
        self.num_failed_agents = self.num_failed_agents + num_unreachable_passengers #record total failed pathfinding agents

        # now lets create the actual passengers
        for j in range(num_nodes): #go through all the nodes we are ending up at
            num_passengers = num_passengers_created[j]
            if num_passengers>0:
                end_node = self.nodes[j]
                path = copy.deepcopy(path_to_nodes[j])
                new_agent = a.Agent(start_node,end_node,self.agent_id_counter,self.time,self,num_passengers,path) #create the new passenger
                self.agents.append(new_agent) #create the new passengers and add to the list
                self.agent_ids.append(self.agent_id_counter) #store the id of the newly created passenger
                self.agent_id_counter = self.agent_id_counter + 1 #increment the id counter
                #assign the passenger to their starting station
                start_node.add_agent(new_agent)
```

```

    #create new passengers at stations, going between each node pair
    def create_all_passengers(self):
        num_nodes = len(self.node_names)
        for i in range(num_nodes): #go through all the nodes we are starting from
            start_node = self.nodes[i] #extract a reference to that node
            for j in range(num_nodes): #go through all the nodes we are ending up at
                end_node = self.nodes[j] #extract a reference to that node
                num_passengers_pair = self.origin_destination_trips[i,j] #extract number
                     of passengers going between these node pairs
                num_passengers_per_min = (num_passengers_pair/60)*self.
                    passenger_time_multipler #we create passengers every minute, but
                    statistics are per day
                self.create_passengers_pair(start_node,end_node,num_passengers_per_min)

    #create the passengers for a specific pair of nodes and edges
    def create_passengers_pair(self,start_node,end_node,num_passengers_per_min):
        int_num_passengers = int(num_passengers_per_min) #rounded-down number of
            passengers to create
        chance_additional_passenger = num_passengers_per_min-int_num_passengers #chance
            of an additional passenger being created from the remainder
        num_passengers = int_num_passengers + random_true(chance_additional_passenger) #
            get the final number of passengers to be created
        #now create the actual passengers at the stations
        if num_passengers>0:
            self.create_passenger(start_node,end_node,num_passengers)


    #create a single passenger
    def create_passenger(self,start_node,end_node,num_passengers):
        #create the passenger
        new_agent = a.Agent(start_node,end_node,self.agent_id_counter,self.time,self,
            num_passengers)
        if new_agent.found_path == True:
            #create the new passenger if they can find a path to their destination
            self.agents.append(new_agent) #create the new passengers and add to the list
```

```
            self.agent_ids.append(self.agent_id_counter) #store the id of the newly created passenger
            self.agent_id_counter = self.agent_id_counter + 1 #increment the id counter
            #assign the passenger to their starting station
            start_node.add_agent(new_agent)
            self.num_successful_agents = self.num_successful_agents + num_passengers
        else:
            #if we cannot find a path to their destination, uncreate the agent
            self.num_failed_agents = self.num_failed_agents + num_passengers



    #update when the next vehicle in each schedule will arrive at each node
    def update_nodes_next_vehicle(self):
        for node in self.nodes:
            #determine when the next service of each schedule will arrive
            node.self_time_till_next_vehicles(self.time)
            #remove services which have already arrived
            node.remove_arrived_vehicles(self.time)


    #passengers alight from vehicles which have stopped
    def alight_passengers(self):
        #loop through all vehicles
        for i,vehicle in enumerate(self.vehicles):
             #if a vehicle is at stop, passengers may alight
            if vehicle.state == 'at_stop':
                stop_node = vehicle.previous_stop #where did the vehicle stop
                #stop_node.next_vehicle_changed = True #the next vehicle stopping at this node will now be different (I don't think this is needed for alighting)
                schedule_name = vehicle.schedule_name
                copy_vehicle_agents = copy.copy(vehicle.agents) #create a shallow copy of the list of agents at the vehicle (agents will be the same, but references will be independent
                num_removed = 0 #keep of number removed so we can pop the right agents
                #go through all the agents on the vehicle
```

```
                for j,agent in enumerate(copy_vehicle_agents):
                    alight_status = agent.alight(stop_node.name)
                    if agent.done==True: #we will not waste our time processing agents that have reached their destination
                        pass
                    else:
                        if alight_status == 1: #agent is alighting
                            agent = vehicle.alight_agent(j-num_removed) #remove them from the list of agents at the vehicle
                           # print('type ',type(agent),' name',agent.name) #DEBUG
                            num_removed = num_removed + 1
                            stop_node.add_agent(agent) #and add them to list of agents at the station
                        elif alight_status == 2: #agent is alighting at their destination
                            agent = vehicle.alight_agent(j-num_removed) #remove them from the list of agents at the vehicle
                          #  print('type ',type(agent),' name',agent.name) #DEBUG
                            num_removed = num_removed + 1
                            agent.done = True  #mark the agent as having achieved their goals
                        elif alight_status == 0: #agent is not alighting
                            pass

    #passengers board vehicles which have stopped
    def board_passengers(self):
        #loop through all vehicles
        for i,vehicle in enumerate(self.vehicles):
            if vehicle.state == 'at_stop':
                #if a vehicle is at stop, we need to board passengers
                stop_node = vehicle.previous_stop #where did the vehicle stop
                stop_node.next_vehicle_changed = True #the next vehicle stopping at this node will now be different
                schedule_name = vehicle.schedule_name
                copy_stop_node_agents = copy.copy(stop_node.agents) #create a shallow copy of the list of agents at the node (agents will be the same, but references will be independent)
```

```
                num_removed = 0 #keep of number removed so we can pop the right agent
                for j,agent in enumerate(copy_stop_node_agents): #go through all the
                    agents where the vehicle stopped
                    original_path = copy.deepcopy(agent.destination_path)
                    will_board = agent.board(schedule_name)
                    if will_board == True:
                        #if the agent is getting on the vehicles
                        vehicle_capacity = vehicle.get_capacity()
                        agent_passengers = agent.number_passengers
                        if agent_passengers<=vehicle_capacity:
                            agent = stop_node.remove_agent(j-num_removed) #remove them
                                from the list of agents at the node, making sure to
                                account for the change in the array size due to removed
                                agents
                            vehicle.board_agent(agent) #have the agents board the
                                vehicle
                            num_removed = num_removed + 1 #we have removed another agent
                        elif vehicle_capacity==0:
                            agent.destination_path = original_path
                        else:
                            leftover_passengers = agent_passengers-vehicle_capacity
                            agent = stop_node.agents[j-num_removed]
                            copy_agent = a.Agent(agent.start_node,agent.destination_node
                                ,agent.id,agent.start_time,agent.network,
                                vehicle_capacity,copy.deepcopy(agent.destination_path))
                            agent.num_passengers = leftover_passengers
                            agent.destination_path = original_path
                            vehicle.board_agent(copy_agent)
                    else:
                        #if agent is not boarding, we do not need to do anything
                        pass

    #update time by one unit
    def update_time(self):
        self.update_passenger_time_multiplier()
        if self.verbose>=1:
            print('time ', self.time)
```

```
        if self.verbose>=1:
            print('at start num passengers ', len(self.agents))
        self.move_vehicles() #move vehicles around the network
        self.update_nodes_next_vehicle() #update when the next vehicles will arrive at each node
        self.alight_passengers() #passengers alight from vehicles
        if self.verbose>=1:
            print('after alighting num passengers ', len(self.agents))
        #self.remove_arrived_vehicles()  #remove vehicles which have completed their path
        self.assign_vehicles_schedule() #create new vehicles at scheduled locations
        self.create_all_passengers_pathfinding() #create new passengers
        if self.verbose>=1:
            print('after creating new, new passengers ', len(self.agents))
        self.board_passengers() #passengers board vehicles
        if self.verbose>=1:
            print('after boarding num passengers ', len(self.agents))
        self.time = self.time + 1 #increment time

    #run for a certain amount of time
    def basic_sim(self):
        self.time = 0
        self.times = []
        final_time = self.stop_simulation_time #determine when the simulation will end
        self.vehicle_logging_init() #initialise vehicle logging
        self.node_logging_init() #initialise node logging
        #create lists to store latitudes,longitudes and names of vehicles over time as lists of lists
        old_real_time = time.time()
        while self.time<final_time:#till we reach the specified time
            self.update_time() #run the simulation
            self.times.append(self.time) #store the current time
            self.get_vehicle_data_at_time() #extract vehicle data at the current time
            self.get_node_data_at_time() #extract node data at the current time
            print("TIME ", self.time,'step took time ',time.time()-old_real_time)
            old_real_time = time.time()
```

```
        print("number of passengers who could reach their destination ",self.
            num_successful_agents)
        print("number of passengers who failed to reach their destination ",self.
            num_failed_agents)
        return self.times, self.vehicle_latitudes, self.vehicle_longitudes, self.
            store_vehicle_names, self.vehicle_passengers, self.node_passengers, self.
            num_failed_agents, self.num_successful_agents, final_time #return relevant
            data from the simulation to the calling code

    #class to initialise class variables to store data about the vehicles as lists of
        lists
    def vehicle_logging_init(self):
        self.vehicle_latitudes = []
        self.vehicle_longitudes = []
        self.store_vehicle_names = []
        self.vehicle_passengers = []

    #class to initalise class variables to store data about nodes as lists of lists
    def node_logging_init(self):
        self.node_passengers = []

    #get relevant data about all vehicles in the network at the present time and store
        them in lists
    def get_vehicle_data_at_time(self):
        current_vehicle_latitudes = []
        current_vehicle_longitudes = []
        current_vehicle_names = []
        current_vehicle_passenger_counts = []
        for vehicle in self.vehicles:
            #extract and store the data at the current time in a list
            latitude,longitude = vehicle.get_coordinates() #get the latitude, longitude
                and direction of the vehicle
            current_vehicle_latitudes.append(latitude)
            current_vehicle_longitudes.append(longitude)
            current_vehicle_names.append(vehicle.name)
            current_vehicle_passenger_counts.append(vehicle.count_agents())
            if self.verbose>=1:
```

```
                print('vehicle ',vehicle.name) #DEBUG
                print('num passengers ',vehicle.count_agents())
            #print(longitude) #DEBUG
        #and store that list in a list containing data for all time
        self.vehicle_latitudes.append(current_vehicle_latitudes)
        self.vehicle_longitudes.append(current_vehicle_longitudes)
        self.store_vehicle_names.append(current_vehicle_names)
        self.vehicle_passengers.append(current_vehicle_passenger_counts)

    #get relevant data about all nodes in the network at the present time and store them
        in lists
    def get_node_data_at_time(self):
        current_node_passenger_counts = []
        for node in self.nodes:
            current_node_passenger_counts.append(node.count_agents())
        self.node_passengers.append(current_node_passenger_counts)

    #call the correct schedule generation code based on the mode we are using
    def create_schedules(self):
        if self.schedule_type == "simple":
            self.create_schedules_simple()
        elif self.schedule_type == "complex":
            self.create_schedules_complex()
        else:
            print(self.schedule_type,' is not a valid schedule type')

    #create the schedule and functionality needed for scheduling using the complex
        method
    #this method constructs the schedules out of "segments", which describe a relatively
        short route through a network
    def create_schedules_complex(self):
        #extract info about the segments
        segment_routes = self.segment_csv["Route"].to_list() #extract the name of the
            route (start-end)
        segment_modifiers = self.segment_csv["Modifier"].to_list() #extract the modifier
            of the route description(eg, fast, semi-fast)
```

```
        segment_txt_schedules = self.segment_csv["Schedule"].to_list() #extract the actual schedule text of the segment
        segment_reverse_txt_schedules = [] #reverse schedule txts
        segment_names = [] #names of the segments
        segment_reverse_names = [] #names of the reverse segments
        num_segments = len(segment_routes) #how many segments are there
        #calculate names and schedules of segments and their reverses
        for i in range(num_segments):
            segment_reverse_txt_schedules.append(reverse_schedule_list_txt(segment_txt_schedules[i])) #determine the reverse schedule
            #determine names of segments
            if segment_modifiers[i]=="":
                new_segment_name = segment_routes[i]
                reverse_segment_name = reverse_segment_route(segment_routes[i])
            else:
                new_segment_name = segment_routes[i]+ ' ' + segment_modifiers[i]
                reverse_segment_name = reverse_segment_route(segment_routes[i]) + ' ' + segment_modifiers[i]
            #add these to the list of segment names
            segment_names.append(new_segment_name)
            segment_reverse_names.append(reverse_segment_name)
        #merge regular and reverse list
        segment_names = segment_names + segment_reverse_names
        segment_txt_schedules = segment_txt_schedules + segment_reverse_txt_schedules
        #extract node names from the segments
        all_segment_nodes = []
        for i in range(num_segments*2):
            segment_nodes = extract_schedule_list_txt(segment_txt_schedules[i])
            all_segment_nodes.append(segment_nodes)

        #now that we have determined the nodes making up a segment
        #we need to combine the segments into schedules
        self.schedule_names = self.schedule_csv["Name"].to_list() #extract the name of schedules (a route that a vehicle will perform)
        self.schedule_gaps = np.array(self.schedule_csv["Gap"].to_list()) #extract the gap in time (in minutes) between services along a particular route
```

```
        self.schedule_offsets = np.array(self.schedule_csv["Offset"].to_list()) #extract
             the offset from the start of time (in minutes) and when the first service
            occurs
        self.schedule_finish = np.array(self.schedule_csv["Finish"].to_list()) #time at
            which the last service of a schedule may depart
        schedule_segments_texts = self.schedule_csv["Schedule Segments"].to_list() #
            extract the raw text that makes up a schedule as a list of segmentss
        self.schedules = [] #list to store schedule objects
        schedule_strings = [] #list of schedule strings in the simple format
        num_schedules = len(self.schedule_names)
        for i in range(num_schedules):
            #for each schedule, extract the segments of the schedule
            segments_in_schedule = extract_schedule_list_txt(schedule_segments_texts[i])
                 #we can reuse this function as it extracts any comma seperated valued
                list
            num_segments = len(segments_in_schedule)
            first_segment = True
            for j in range(num_segments):
                try:
                    segment_id =  segment_names.index(segments_in_schedule[j])
                except:
                    print('error cannot find "',segments_in_schedule[j], '" in list of
                        segment names')
                else:
                    #if we can find the segment ids
                    segment_nodes = copy.deepcopy(all_segment_nodes[segment_id]) #copy
                        to prevent modifying originals
                    if first_segment==True:
                        #initial list of nodes is just the segment nodes
                        nodes = segment_nodes
                        first_segment = False #
                    else:
                        last_node_previous = nodes[-1] #get the last node of the
                            previous segment
                        first_node_new = segment_nodes[0] #get the first node of the new
                             segment
```

```
                        if first_node_new==last_node_previous: #this too must match otherwise the schedule is invalid
                            nodes.pop() #remove last node from the previous segment
                            nodes = nodes + segment_nodes #add the nodes from the new segment
                        else:
                            #DEBUG
                            print('last node of schedule "',segments_in_schedule[j-1],'" "',last_node_previous, '" does not match first node of schedule "',segments_in_schedule[j],'" "',first_node_new,'"')
                            print('hence schedule "',self.schedule_names[i], '" is invalid')

                #once we have extracted the list of nodes
                schedule_string = make_schedule_string(nodes) #convert back into a schedule string
                schedule_strings.append(schedule_string) #and store
            #now create the actual schedule objects
            for i in range(num_schedules):
                self.schedules.append(self.create_schedule(self.schedule_names[i], schedule_strings[i])) #create a schedule object for each schedule
            #create the dispatch schedule

        def create_dispatch_schedule(self):
            num_schedules = len(self.schedule_names)
            self.dispatch_schedule2 = []
            for i in range(num_schedules):
                #create the dispatch schedule for each particular schedule
                single_dispatch_schedule = []
                service_time = self.schedule_offsets[i] #extract the starting time of a service
                finish_time = self.schedule_finish[i] #and the last time at which a service can start
                service_gap = self.schedule_gaps[i]
                while service_time<=finish_time:
```

```
                single_dispatch_schedule.append(service_time) #add the time of the
                    service to the dispatch schedule
                service_time = service_time + service_gap #calculate when the next
                    service will occur
            #once we have added all the departure times for this service, store it in
                the overall dispatch schedules
            self.dispatch_schedule2.append(single_dispatch_schedule)


    #create the schedule and functionality needed for scheduling using the simple method
    def create_schedules_simple(self):
        self.schedule_names = self.schedule_csv["Name"].to_list() #extract the name of
            schedules (a route that a vehicle will perform)
        self.schedule_gaps = np.array(self.schedule_csv["Gap"].to_list()) #extract the
            gap in time (in minutes) between services along a particular route
        self.schedule_offsets = np.array(self.schedule_csv["Offset"].to_list()) #extract
             the offset from the start of time (in minutes) and when the first service
            occurs
        self.schedule_finish = np.array(self.schedule_csv["Finish"].to_list())
        schedule_texts = self.schedule_csv["Schedule"].to_list() #extract the raw text
            that makes up a schedule
        self.schedules = [] #list to store the schedule objects
        num_schedules = len(self.schedule_names)

        for i in range(num_schedules):
            self.schedules.append(self.create_schedule(self.schedule_names[i],
                schedule_texts[i])) #create a schedule object for each schedule


    #create a schedule object from a name and a text string
    def create_schedule(self,name,schedule_string):
        node_names = extract_schedule_list_txt(schedule_string) #extract node names from
             the schedule string
        num_nodes = len(node_names)
        node_arrival_times = np.zeros(num_nodes)#arrival times at each node, starting
            from 0 at the starting node
        node_counter = 0 #which node is currently the next destination
```

```
        new_schedule = schedule.Schedule(name)
        previous_node_name = ""
        #add nodes and edges to the schedule
        for node_name in node_names:
            #when processing the starting node, we just add the node to the schedule
            node = self.nodes[self.get_node_index(node_name)]
            if previous_node_name == "":
                new_schedule.add_start_node(node,node_name)
                previous_node = node
                previous_node_name = node_name
                node_counter += 1 #we will now be processing the next node
            else:
                edge_name = previous_node_name + ' to ' + node_name #calculate the name
                    of the edge between these two nodes
                edge = self.edges[self.get_edge_index(edge_name)]
                edge_time = edge.provide_travel_time()
                new_schedule.add_destination(node,edge,node_name)
                node_arrival_times[node_counter] = node_arrival_times[node_counter-1] +
                    edge_time
                previous_node = node
                previous_node_name = node_name
                node_counter += 1

        #now store arrivial times in the schedule
        new_schedule.add_schedule_times(node_arrival_times)
        return new_schedule

    #determine which nodes have which schedules present
    def determine_which_nodes_have_schedule(self):
        #go through all the nodes
        for i,node_name in enumerate(self.node_names):
            #find all the nodes
            for j,schedule in enumerate(self.schedules):
                #find out if that node name is in that schedule and if so return nodes
                node_found,search_node_time,nodes_after,node_times_after = schedule.
                    node_name_in_schedule(node_name)
                if node_found == True:
```

```
                    #if we found the node in a schedule, add that schedule to the list of schedules stopping at that node
                    self.nodes[i].add_stopping_schedule(self.schedule_names[j],self.dispatch_schedule2[j],search_node_time,nodes_after,node_times_after)

    #add an edge between specified start and end node
    def add_edge(self,start_node,end_node,travel_time):
        name = start_node + ' to ' + end_node
        while name in self.edge_names:#prevent duplicate names
            #note, that duplicate edge names cause problems with the creation of schedules, so try and avoid them
            warnings.warn('duplicate edge name ', name, ' this is poorly supported, try and only have one edge directly between two nodes')
            name = name + ' alt '
        self.edge_names.append(name)#update the list of edge names
        new_edge = Edge(name,start_node,end_node,travel_time)
        self.edges.append(new_edge)#and create the new edge
        #let's also add the edge to the list of edges at the node it starts from
        i = 0 #temporary counter variable
        for node_name in self.node_names:
            if node_name == start_node:#if node names matches with the starting node
                #add edge to the node
                self.nodes[i].add_edge(new_edge)
            else:
                pass
            i = i+1
            #increment the counter
            pass

    #find the time taken to travel from the specified node to all other nodes in the network
    #note, this is making the assumption that all nodes are always traversible, the ideal case which does not apply for real passengers
    def find_distance_dijistraka(self,start_node_name):
        #try and find the starting node in the list of all nodes
        try:
```

```
            start_index = self.node_names.index(start_node_name)
        except ValueError:
            #handle case where starting name not in list of names
            warnings.warn('start_node_name ', start_node_name, 'is not in the list of node names in this network')
            return False #return false to indicate error
        #if there was not an error, continue
        num_nodes = len(self.node_names)
        distance_to_nodes = np.ones(num_nodes)*np.inf #set initial cost to reach to be infinite, index order is same as in node names
        nodes_visited = np.zeros(num_nodes) #has node been visited yet, 0 if false, infinite if true
        distance_to_nodes[start_index] = 0 #cost to reach starting node is of course zero
        while True:
            distance_to_use = distance_to_nodes + nodes_visited#consider the cost to reach already visited nodes to be infinite, to prevent the need to look at them twice
            min_distance = np.min(distance_to_use)#get the minimum distance in the array
            if min_distance == np.inf: #if all nodes are either visited or have an infinite known cost to reach, we have explored the network as much as possible
                break#hence break
            min_index = distance_to_use.tolist().index(min_distance)#get the index of the first minimum value
            (edge_times, edge_destinations) = self.nodes[min_index].provide_nodes_time()
            num_edges = len(edge_times)
            for i in range(num_edges):
                try:
                    destination_index = self.node_names.index(edge_destinations[i])
                except ValueError:
                    #handle case where destination name not in list of names
                    print('WARNING destination name ', edge_destinations[i], 'is not in the list of node names in this network')
                    continue #skip remaining computation steps

```

```
                new_distance = min_distance + edge_times[i] #calculate distance to reach destination through the current node
                if new_distance < distance_to_nodes[destination_index]:#if distance through current node is less than the current minimum distance
                    distance_to_nodes[destination_index] = new_distance #update the distance
            #now we have looked at this node, update the nodes we have visited
            nodes_visited[min_index] = np.inf #indicate we have visited the node

        return distance_to_nodes #return the distance to all the nodes

    #this is the same as find_distance_dijistraka, but it also stores the path as a list of nodes
    def find_distance_dijistraka_path(self,start_node_name):
        #try and find the starting node in the list of all nodes
        try:
            start_index = self.node_names.index(start_node_name)
        except ValueError:
            #handle case where starting name not in list of names
            print('WARNING start_node_name ', start_node_name, 'is not in the list of node names in this network')
            return False #return false to indicate error
        #if there was not an error, continue
        num_nodes = len(self.node_names)
        distance_to_nodes = np.ones(num_nodes)*np.inf #set initial cost to reach to be infinite, index order is same as in node names
        #create paths array, this will be a list of paths, with each path a list of edges
        paths = [[] for _ in range(num_nodes)]
        nodes_visited = np.zeros(num_nodes) #has node been visited yet, 0 if false, infinite if true
        distance_to_nodes[start_index] = 0 #cost to reach starting node is of course zero
        while True:
            distance_to_use = distance_to_nodes + nodes_visited#consider the cost to reach already visited nodes to be infinite, to prevent the need to look at them twice
```

```
            min_distance = np.min(distance_to_use)#get the minimum distance in the array to a node we know how to reach
            if min_distance == np.inf: #if all nodes are either visited or have an infinite known cost to reach, we have explored the network as much as possible
                break#hence break
            min_index = distance_to_use.tolist().index(min_distance)#get the index of the first minimum value
            (edge_times,edge_destinations,edge_names) = self.nodes[min_index].provide_nodes_time_edge_name()
            num_edges = len(edge_times)
            for i in range(num_edges):
                try:
                    destination_index = self.node_names.index(edge_destinations[i])
                except ValueError:
                    #handle case where destination name not in list of names
                    print('WARNING destination name', edge_destinations[i], 'is not in the list of node names in this network')
                    continue #skip remaining computation steps

                new_distance = min_distance + edge_times[i] #calculate distance to reach destination through the current node
                if new_distance < distance_to_nodes[destination_index]:#if distance through current node is less than the current minimum distance
                    distance_to_nodes[destination_index] = new_distance #update the distance
                    minimum_path = paths[min_index].copy()
                    minimum_path.append(edge_names[i])  #add the new edge to the minimum path to start node to get the minimum path to the end node
                    paths[destination_index] = minimum_path #store the shortest path to the new node

            #now we have looked at this node, update the nodes we have visited
            nodes_visited[min_index] =  np.inf #indicate we have visited the node

        return distance_to_nodes,paths #return the distance to all the nodes

```

```
    #find the distance to travel to all nodes from all nodes
    def find_distance_to_all(self):
        num_nodes = len(self.node_names)
        distance_arrays = [] #list to store distance arrays from a particular node
        #generate the distance arrays from each node
        for i in range(num_nodes):
            distance_arrays.append(self.find_distance_dijistraka(self.node_names[i]))
        #and merge them into a numpy array

        self.distance_to_all = np.stack(distance_arrays)
        return self.distance_to_all

    #as above, but also store the routes taken
    def find_distance_to_all_path(self):
        num_nodes = len(self.node_names)
        distance_arrays = [] #list to store distance arrays from a particular node
        path_arrays = [] #list to store path lists from each node
        #generate the distance arrays from each node
        for i in range(num_nodes):
            new_distance,new_paths = (self.find_distance_dijistraka_path(self.node_names
                [i]))
            distance_arrays.append(new_distance)
            path_arrays.append(new_paths)

        #and merge them into a numpy array
        self.distance_to_all = np.stack(distance_arrays)
        self.paths_to_all = path_arrays
        return self.distance_to_all

    #find the expected traffic along each edge in each direction
    def find_expected_edge_traffic(self):
        #create the array
        num_edges = len(self.edge_names)
        self.edge_traffic = np.zeros(num_edges)
        #go through all the shortest path between node_pairs
        for outer_index,paths in enumerate(self.paths_to_all):
            for inner_index,path in enumerate(paths):
```

```
                #extract the amount of traffic along the path between the selected nodes
                node_to_node_traffic = self.origin_destination_trips[outer_index,
                    inner_index]
                for edge_name in path:#go through all the edge names in the path
                    edge_index = self.get_edge_index(edge_name) #find the index of the
                        edge we are pathing through
                    self.edge_traffic[edge_index] = self.edge_traffic[edge_index] +
                        node_to_node_traffic #add the traffic from the new edge

    #create a matrix of travel demand between each node using the gravity model
    def create_origin_destination_matrix(self):
        num_passengers = np.array(self.node_passengers)
        #use gravity model with 1D distance dropoff and 5 minute flat distance (these
            fudge factors are decided because they produce good results)
        self.origin_destination_trips = gravity_assignment(starts=num_passengers,stops=
            num_passengers,distances=self.distance_to_all,distance_exponent=1,
            flat_distance=5,verbose=self.verbose)
        return self.origin_destination_trips

    #get the index of a node name in the list of nodes
    def get_node_index(self,node_name):
        #try and find the starting node in the list of all nodes
        try:
            index = self.node_names.index(node_name)
            return index
        except ValueError:
            #handle case where starting name not in list of names
            print('node_name ', node_name, 'is not in the list of node names in this
                network')
            return -1 #return -1 to indicate error

    #get the index of an edge name in the list of edges
    def get_edge_index(self,edge_name):
        #try and find the starting node in the list of all nodes
        try:
            index = self.edge_names.index(edge_name)
            return index
```

```
        except ValueError:
            #handle case where starting name not in list of names
            print('edge_name  ', edge_name, 'is not in the list of edge names in this
                network')
            return -1 #return -1 to indicate error

    #get the time taken to traverse a node
    def get_edge_time(self,edge_name):
        index = self.get_edge_index(edge_name)
        time_taken = self.edges[index].provide_travel_time()
        return time_taken

    #get the traffic through a node
    def get_edge_traffic(self,edge_name):
        index = self.get_edge_index(edge_name)
        traffic = self.edge_traffic[index]
        return traffic

    #provide a breakdown of where passengers starting at a particular node are going
    def test_origin_destination_matrix(self,start_node_name):
        #try and find the starting node in the list of all nodes
        start_index = self.get_node_index(self,start_node_name)
        if start_index==-1:
            return False
        #if there was not an error, continue
        num_nodes = len(self.node_names)
        trips_from_start = self.origin_destination_trips[start_index:start_index+1,:][0]
        print(trips_from_start)
        num_trips = np.sum(trips_from_start)
        percent_trips = trips_from_start/num_trips
        print('from ',start_node_name,' ',num_trips, ' passengers travel')
        for i in range(num_nodes):
            with np.printoptions(precision=2,suppress=True):
                print(' to ',self.node_names[i],' ', trips_from_start[i], ' passengers
                    which is', percent_trips[i]*100 ,' %')

    def test_origin_destination_matrix_all(self):
```

```
        num_nodes = len(self.node_names)
        stops = np.sum(self.origin_destination_trips,0)
        total_stops = np.sum(stops)
        percent_trips = stops/total_stops
        print('across all nodes, passengers travel')
        for i in range(num_nodes):
            with np.printoptions(precision=2,suppress=True):
                print(' to ',self.node_names[i],' ', stops[i], ' passengers which is',
                    percent_trips[i]*100 ,' %')
            #print(' to ',self.node_names[i],' ', f"{stops[i]:.2f}", ' passengers which
                is', f"{percent_trips[i]*100:.2f}" ,' %')
        for i in range(num_nodes):
            self.test_origin_destination_matrix(self.node_names[i])

    #testing functionality
    def test_nodes(self):
        for i  in range(len(self.nodes)):
            self.nodes[i].test_node()

    def test_edges(self):
        for i in range(len(self.edges)):
            self.edges[i].test_edge()

    def test_dijistraka(self,start_node):
        print('from ', start_node, ' time to reach is ')
        best_distance_to_nodes = self.find_distance_dijistraka(start_node)
        num_nodes = len(self.node_names)
        for i in range(num_nodes):
            print(self.node_names[i], ' time ',best_distance_to_nodes[i])

    def test_schedules(self):
        num_schedules = len(self.schedule_names)
        #test all the schedules in the network
        for i in range(num_schedules):
            self.schedules[i].test_schedule()

    def test_verbose(self):
```

```
        print('verbosity = ',self.verbose)
        if self.verbose==0:
            print('verbosity is 0')
        if self.verbose>=1:
            print('verbosity is greater or equal to 1')
        if self.verbose>=2:
            print('verbosity is greater or equal to 2')


#assign trips between origin destination pairs using the gravity model
#starts/stops are number of passengers starting/stopping at particular nodes (1D Numpy array)
#distances is amount of time taken (in ideal world) to travel between each pair of nodes (2D Numpy array)
#length of all these arrays MUST be equal
#distance exponent is how much cost scales with distance
#flat distance is default amount of distance applied on top to all trips
#iterations is how many iterations to converge
#as yet unsure how well this handles
def gravity_assignment(starts,stops,distances,distance_exponent,flat_distance,verbose=1,required_accuracy=0.001,max_iterations=100):
    distances = (distances+flat_distance)**distance_exponent #calculate distance after transforms
    num_nodes = len(starts)
    destination_importance_factors = np.ones(num_nodes)#correction factor used to ensure convergence of number of trips to a node with recorded number of stops at that node
    list_trips = [] #list to store the number of trips pending conversion to a numpy array
    for j in range(num_nodes):#go through all the starting nodes
        this_node_starts = starts[j]#record the number of trips starting at a node
        trip_importance = np.zeros(num_nodes)#importance of trips to each node from this node
        for k in range(num_nodes):#go through each destination from all nodes
            if k==j:#don't evaluate number of trips from a node to itself
                continue
            else:
```

```
                distance_between = (distances[k,j]+distances[j,k]) #use the round-trip distance, as most passengers intend to return to their origin so this is what determines expected cost of the trip

                trip_importance[k] = ((destination_importance_factors[k]*stops[k])/distance_between)

        num_trips = (trip_importance/np.sum(trip_importance))*this_node_starts #calculate the number of trips from this node to all other nodes
        list_trips.append(num_trips)

    calc_trips = np.stack(list_trips)#merge the number of trips from each node to each destination into a numpy array
    iter = 0
    while True:
        calc_stops = np.sum(calc_trips,0)
        calc_starts = np.sum(calc_trips,1)
        stop_correction_factor = stops/calc_stops
        start_correction_factor = starts/calc_starts
        abs_start_error = np.abs(start_correction_factor-1)
        abs_stop_error = np.abs(stop_correction_factor-1)
        if (max(abs_stop_error)<required_accuracy) and (max(abs_start_error)<required_accuracy):
            if verbose>=1:
                print("desired accuracy achieved after ", iter, " iterations")
            break
        elif iter>=max_iterations:
            if verbose>=1:
                print("failed to converge after ",max_iterations," iterations")
            break
        else:
            iter = iter+1
        #now apply the stop correction factor to traffic
        for j in range(num_nodes):#go through starting node
            for k in range(num_nodes):#go through destination node
```

```
                calc_trips[j,k] = calc_trips[j,k]*stop_correction_factor[k] #multiply
                    the number of trips going to each destination node by the stop
                    correction factor of that destination
        calc_stops = np.sum(calc_trips,0)
        calc_starts = np.sum(calc_trips,1)
        start_correction_factor = starts/calc_starts
        #print('start correction factors ',start_correction_factor)
        #now apply the start correction factor to traffic
        for j in range(num_nodes):#go through starting node
            for k in range(num_nodes):#go through destination node
                calc_trips[j,k] = calc_trips[j,k]*start_correction_factor[j]  #multiply
                    the number of trips from each origin by the start correction factor
                    of that origin


        #print('after start calibration')

    if verbose>=2:
        print('at the end')
        calc_stops = np.sum(calc_trips,0)
        print('calc stops ',calc_stops)
        stop_error = (calc_stops/stops)
        print('stop correctness ',stop_error)
        calc_starts = np.sum(calc_trips,1)
        print('calc starts ',calc_starts)
        start_error = calc_starts/starts
        print('start correctness ',start_error)
        print('biggest errors rates are')
        abs_start_error = np.abs(start_error-1)
        abs_stop_error = np.abs(stop_error-1)
        print('for start, max error ', np.max(abs_start_error),' mean error ',np.mean(
            abs_start_error))
        print('for stop, max error ', np.max(abs_stop_error),' mean error ',np.mean(
            abs_stop_error))
        print('end testing')
    return calc_trips

```

```
#extract latitude and longitude from a string of coordinates (in the format provided by
    google maps)
def extract_coordinates(coordinates):
    #extract the latitude and longitude strings
    latitude = ''
    longitude = ''
    extracting_longitude = False
    i = 0
    while i < len(coordinates):
        if coordinates[i] == ',':
            extracting_longitude = True
            i = i + 2
        else:
            if extracting_longitude:
                longitude += coordinates[i]
            else:
                latitude += coordinates[i]
            i = i + 1

    return float(latitude),float(longitude)

#extract a list of nodes in a schedule from a text string
def extract_schedule_list_txt(schedule_string):
    new_node = []
    nodes = []
    for letter in schedule_string:
        if letter==',': #move onto the next node when the delimiter is reached
            nodes.append("".join(new_node))#append the node name to the list of nodes
            new_node = [] #reset the node
        else:
            new_node.append(letter) #append the letter to the node name
    #also add on the final node (after the last comma)
    nodes.append("".join(new_node))
    return nodes

#reverse the order of nodes in a schedule string
```

```
def reverse_schedule_list_txt(schedule_string):
    nodes = extract_schedule_list_txt(schedule_string) #get the list of node names
    nodes.reverse() #reverse the list of nodes
    #reconvert it back into a text string
    schedule_string = ""
    for node in nodes:
        schedule_string = schedule_string + node + ','
    #remove the trailing comma
    schedule_string = schedule_string[:-1]
    return schedule_string

#reverse the route name of a segment
def reverse_segment_route(route_name_string):
    start_node_name = ""
    end_node_name = ""
    start_node_extracted = False
    for letter in route_name_string:
        if start_node_extracted==False:
            if letter=='-':
                start_node_extracted = True
            else:
                start_node_name = start_node_name + letter
        else:
            end_node_name = end_node_name + letter

    reverse_name = end_node_name + "-" + start_node_name
    return reverse_name


#return true if random generated number is less than provided chance
#input chance is equal to the chance of the output being true
def random_true(chance):
    random_number = rand.random() #random number between 0 and 1
    if random_number<=chance:
        return True
    else:
        return False
```

```python
#turn a list of nodes into a schedule string
def make_schedule_string(nodes):
    schedule_string = ""
    for node in nodes:
        #add each node name to the schedule string
        schedule_string = schedule_string + node + ','
    schedule_string = schedule_string[:-1] #remove trailing comma
    return schedule_string
```

A1.3 agent.py

```python
import numpy as np
import copy as copy
#agent.py
#stores the agent class and related functionality

#route_step = [next_service_name,node.name]

class Agent:
    def __init__(self,start_node,destination_node,id,start_time,network,
        number_passengers,path):
        self.start_node = start_node
        self.destination_node = destination_node
        self.id = id
        self.start_time = start_time
        self.network = network #reference to the network object
        self.destination_path = path #path of actions to the destination node
        self.number_passengers = number_passengers #number of passengers represented by
            this agent
        #self.found_path = self.pathfind()
        self.done = False #has the agent reached their destination yet


    #calculate a path from the start to the destination
    #store this path inside the agent
    def pathfind(self):
```

```
        #print('start ',self.start_node.name,' destination ',self.destination_node.name) #DEBUG
        #get info about vehicles arriving at the starting node
        start_next_service_times, start_nodes_after, start_node_times_after, start_schedule_names = self.start_node.provide_next_services(data_time=self.start_time, start=True)
        #get index (id) of starting and ending nodes in the network structure
        start_node_index = self.start_node.id
        destination_node_index = self.destination_node.id
        #create an array to store the paths to all the other nodes
        num_nodes_in_network = len(self.network.node_names)
        distance_to_nodes = np.zeros(num_nodes_in_network) + np.inf #initial distance to reach all other nodes will be infinite
        evaluated_nodes = np.zeros(num_nodes_in_network) #when a node is evaluated the value in this matrix is set to infinite, ensuring that node is never evaluated again
        distance_to_nodes[start_node_index] = 0 #initial distance to reach the starting node is 0
        distance_to_final_destination = self.network.distance_to_all[:, destination_node_index]
        path_to_nodes = [[] for _ in range(num_nodes_in_network)] #create an empty nested list of the required length to store paths to nodes
        #now that we have extracted preliminary data, start the pathfinding operation
        while True: #loop till we meet an exit condition
            expected_distance_to_nodes = distance_to_nodes + distance_to_final_destination + evaluated_nodes #expected (minimal) distance to reach a node
            min_index = np.argmin(expected_distance_to_nodes) #get the index of the node with the lowest expected travel time, evaluate this next
            minimum_expected_distance = expected_distance_to_nodes[min_index]
            #print('evaluating ',self.network.nodes[min_index].name,' which takes ', distance_to_nodes[min_index],' to reach from start') #DEBUG
            #print('and ',expected_distance_to_nodes[min_index],' to reach final through ') #DEBUG
            if minimum_expected_distance == np.inf:
                break #break out of the loop, we have explored all the network we can reach
```

```
        elif min_index == destination_node_index:
            #print('we have found the destination node')
            self.destination_path = path_to_nodes[destination_node_index]
            #print(self.destination_path)
            break
        else:
            minimum_distance = distance_to_nodes[min_index] #extract the time taken to reach the node being evaluated
            current_time = minimum_distance + self.start_time #time at which we reach the node currently being evaluated
            #otherwise, explore paths from the minimal node
            if min_index==start_node_index:
                #use precalculated data from the starting node
                next_service_times = start_next_service_times
                nodes_after = start_nodes_after
                times_after = start_node_times_after
                schedule_names = start_schedule_names

            else:
                #otherwise calculate data about vehicle arrivials at nodes on the fly
                next_service_times,nodes_after,times_after,schedule_names = self.network.nodes[min_index].provide_next_services(start=False,data_time=current_time)

            #now it's time to calculate the path to other nodes
            num_schedules = len(next_service_times)
            for i in range(num_schedules):
                #extract nodes and times after for this specific route
                next_service_time = next_service_times[i]
                next_service_name = schedule_names[i]
                route_nodes_after = nodes_after[i]
                route_times_after = times_after[i]
                for j,node in enumerate(route_nodes_after):
                    node_index = node.id
```

```
                    distance_to_current_node_old_path = distance_to_nodes[node_index] #what is the current shortest path to the node we are looking at
                    distance_to_current_node_new_path = minimum_distance + (next_service_time-current_time) + route_times_after[j] #how long to reach next node through evaluation node
                    #print('to reach ',node.name,' current best is ', distance_to_current_node_old_path,' new path is ', distance_to_current_node_new_path) #DEBUG
                    if distance_to_current_node_new_path < distance_to_current_node_old_path:
                        #if so, we have found a better path
                        #print('we have found a better path') #DEBUG
                        distance_to_nodes[node_index] = distance_to_current_node_new_path
                        route_to_old_node = path_to_nodes[min_index] #extract the path to the evaluation node
                        #print('route to previous node ',route_to_old_node) #DEBUG
                        #print('route step ',route_step)
                        route_to_new_node = copy.copy(route_to_old_node) #path to the next node is path to the evaluation node + new step
                        route_to_new_node.append(next_service_name) #store the next service we need to catch
                        route_to_new_node.append(node.name) #and when we need to get off that service
                        #print('new route ',route_to_new_node) #DEBUG
                        path_to_nodes[node_index] = route_to_new_node #store this in the list of all paths

            #mark the evaluated node as evaluated, it will not be evaluated again
            evaluated_nodes[min_index] = np.inf

    if distance_to_nodes[destination_node_index]==np.inf: #we have not found a path to our destination
        #hence the passenger should pop back out of existance
        return False #the passenger did not find a path to their destination
    else:
```

```
            return True #indicate we successfully found a path to their destination
    #ask the agent if it wishes to board a vehicle of a particular schedule
    def board(self,schedule_name):
        #print('boarding' ,self.destination_path)
        if schedule_name==self.destination_path[0]:
            #print('boarding boarding')
            #board if schedule name matches with next schedule to board
            del self.destination_path[0] #we only wish to board this service once
            #print('boarding',self.destination_path)
            return True
        else:
            return False

    #ask the agent if it wishes to alight a vehicle at a particular node
    def alight(self,node_name):
        #print('alighting',self.destination_path)
        #print('node name ',node_name)
        if node_name==self.destination_path[0]:
            #print('alighting alighting')
            #alight if node name matches with next node to alight at
            del self.destination_path[0] #we only wish to alight at this node once
            #print('alighting',self.destination_path)
            if len(self.destination_path)==0:
                return 2 #indicate agent has come to the end of its journey after alighting here
            else:
                return 1 #indicate agent has alighted here, but still exists
        else:
            return 0 #indicate not alighting here

    #print the path from the start destination to the end destination
    def test_agent_path(self):
        print('START ',self.start_node.name)
        print('DESTINATION ',self.destination_node.name)
        print("PATH ",self.destination_path)
```

A1.4 vehicle.py

```
#vehicle.py
#stores the vehicle class and related functionality

import copy #for making shallow copies of schedules, we want the schedule object to be unique but the linked nodes/edges to be the same
import schedule as Schedule
import network as Network
#base vehicle class
class Vehicle:
    #create the vehicle
    def __init__(self,schedule,start_time,name,seated_capacity=960,standing_capacity=1680):
        self.schedule = copy.copy(schedule)
        self.schedule_name = self.schedule.name
        self.name = name
        self.state = 'at_stop' #vehicle states are 'at_stop' and 'moving'
        self.state_new = True #newly created, will not stop if final_destination = current destination to allow the city circle to function
        self.schedule.offset_schedule_times(start_time)#adjust the schedule to reflect the time we started
        self.number_passengers = 0 #current number of passengers aboard the vehicle
        check,self.previous_stop = self.schedule.provide_next_destination() #get the starting destination which will be stored as the previous stop
        check = self.schedule.remove_reached_destination() #remove starting destination from list of destinations
        self.final_destination = self.schedule.provide_final_destination() #get the final destination as well
        self.at_final_destination = False #mark if a vehicle has reached it's final destination, and will be deleted next update
        self.agents = [] #container to store agents in the vehicle
        self.num_passengers = 0 #number of passengers in the vehicle
        self.max_passengers = 1610 #maximum number of passengers in the vehicle

    #have an agent try and board the vehicle
    def board_agent(self,agent):
```

```python
            self.agents.append(agent) #add agents to the list of agents on the vehicle
            self.num_passengers = self.num_passengers + agent.number_passengers #the number of passengers has increased

    #have an agent try and leave the vehicle
    def alight_agent(self,id):
        removed_agent = self.agents.pop(id)
        self.num_passengers = self.num_passengers - removed_agent.number_passengers #the number of passengers has decreased
        return removed_agent

    def get_capacity(self):
        return self.max_passengers-self.num_passengers

    #move the vehicle around the network according to its schedule
    def update(self):
        if self.state == 'at_stop': #if the vehicle was at a stop
            #add some code to disembark passengers
            #add some code to pick up passengers
            if self.final_destination == self.previous_stop and self.state_new == False: #if vehicle has reached it's destination and not newly created
                return False #return false to indicate it should be deleted
            #if vehicle has not reached it's final destination
            self.state_new=False
            check,self.next_destination,self.next_edge = self.schedule.provide_next_destination() #extract next destination and how to get there
            self.edge_length = self.next_edge.provide_travel_time() #store the length of the next edge
            if self.edge_length == 1: #if edge takes only 1 time unit to traverse
                #we are immediately at the next destination
                self.state = 'at_stop'
                self.previous_stop = self.next_destination
                self.schedule.remove_reached_destination() #remove the previous destination
            else:
                #we are now moving towards the next destination
```

```
                self.state = 'moving'
                self.move_timer = 1#start the move timer, we will move 1 unit of time

        elif self.state == 'moving': #if the vehicle was moving
            if self.move_timer == self.edge_length-1: # we have reached the next station
                self.state = 'at_stop'
                self.previous_stop = self.next_destination
                self.schedule.remove_reached_destination() # remove the previous destination
            else:
                #we are still moving towards the next destination
                self.state = 'moving'
                self.move_timer = self.move_timer + 1

        return True

    #print where the vehicle is
    def verbose_position(self):
        print('vehicle ',self.name, 'is ',self.state,' path is ',self.schedule_name)
        schedule_nodes = self.schedule.nodes
        for node in schedule_nodes:
            print('too ',node.name)
        #print('currently is ',self.state, 'previous stop is ',self.previous_stop.name,' next stop is ',self.next_destination.name,' move timer is ',self.move_timer)

    #print when the vehicle is at a stop
    def verbose_stop(self):
        if self.state == 'at_stop':
            print('vehicle ',self.name,' stopped at ', self.previous_stop.name)

    def get_coordinates(self):
        if self.state == 'at_stop':
            #when at stop, vehicle position is the position of the stop (which is previous stop)
            latitude = self.previous_stop.latitude
            longitude = self.previous_stop.longitude
```

```
        elif self.state == 'moving':
            #when moving, vehicle position is along straight line path between previous
                node and next node
            fraction_moved = (self.move_timer/self.edge_length)
            latitude = self.previous_stop.latitude*(1-fraction_moved) + (self.
                next_destination.latitude*fraction_moved)
            longitude = self.previous_stop.longitude*(1-fraction_moved) + (self.
                next_destination.longitude*fraction_moved)

        return latitude,longitude


    #count the number of agents in the vehicle
    def count_agents(self):
        #num_agents = 0
        #for agent in self.agents:
        #     num_agents = num_agents + agent.number_passengers
        return self.num_passengers
```

A1.5 schedule.py

```
#schedule.py
#schedule class, stores the list of nodes the vehicle is trying to reach, and the edge
    needed to reach each node
import numpy as np
import copy as copy

class Schedule:
    #initialise the empty schedule
    def __init__(self,name):
        self.name = name#starting node of the schedule, useful for assigning schedules
            to vehicles
        self.nodes = [] #list of destinations (reference to a node)
        self.node_names = [] #list of node names
```

```
        self.edges = [] #list of edges to reach each destination from previous location
            (reference to an edge)
        self.schedule_times = [] #list of times when we will reach the nodes we are
            travelling too

    #create a shallow copy of the object and all it's internal data-structures, however
        maintain same references to nodes and edges
    def __copy__(self):
        #create a schedule object
        copy_schedule = Schedule(self.name)
        copy_schedule.nodes = copy.copy(self.nodes)
        copy_schedule.edges = copy.copy(self.edges)
        copy_schedule.schedule_times = copy.copy(self.schedule_times)
        return copy_schedule

    #add the first destination to the schedule
    def add_start_node(self, start_node, start_node_name):
        self.nodes.append(start_node)
        self.node_names.append(start_node_name)

    #add a destination to the schedule
    def add_destination(self, next_node, next_edge, next_node_name):
        self.nodes.append(next_node)
        self.edges.append(next_edge)
        self.node_names.append(next_node_name)

    #provide final destination in the schedule
    def provide_final_destination(self):
        num_nodes = len(self.nodes)
        final_destination = self.nodes[num_nodes-1]
        return final_destination

    #provide next destination, note this requires you to have first deleted the initial
        destination to work correctly
    def provide_next_destination(self):
        #print('providing next destination, num nodes ',len(self.nodes),' num edges ',
            len(self.edges)) #DEBUG
```

```
        if len(self.nodes)==0:
            return False#return false to indicate there are no more destinations, schedule is finished
        if len(self.nodes)>len(self.edges): #provide the start point if we are yet to remove it
            return (True, self.nodes[0])
        else:
            #return true to indicate there is a next destination, provide next destination and how to get there
            return (True, self.nodes[0], self.edges[0])

    #remove the destination we just reached and the node we used to reach it
    def remove_reached_destination(self):
        if len(self.nodes)==0:
            return False#return false to indicate there are no more destinations, schedule is finished
        if len(self.nodes)>len(self.edges): #remove the start point if we are yet to remove it
            del self.nodes[0]
            return True
        else:
            del self.nodes[0]
            del self.edges[0]
            return True#return true to indicate operation successful

    def provide_name(self):
        return self.name

    def add_schedule_times(self, arrival_times):
        self.schedule_times = arrival_times #this is a numpy array

    #offset the schedule times by the current time to ob'tain time the time the vehicle will reach each node
    def offset_schedule_times(self, current_time):
        self.schedule_times = self.schedule_times + current_time

```

```
    #provide information about the schedule, namely the list of nodes and edges traversed, and the time when nodes will be reached
    def test_schedule(self):
        print('SCHEDULE ', self.name)
        num_nodes = len(self.nodes)
        for i in range(num_nodes):
            if i>0:
                print('NODE ', self.nodes[i].name, ' TIME ', self.schedule_times[i], ' EDGE ', self.edges[i-1].name) #note, print the edge to reach the displayed node
            else:
                print('NODE ', self.nodes[i].name, ' TIME ', self.schedule_times[i]) #for starting node, there is no edge to reach the displayed node

    #find if a node name is in the schedule, and if so, return the nodes after and the times to reach them from the search node
    #also return the time to reach the search node from the start of the schedule
    def node_name_in_schedule(self,search_node_name):
        nodes_after = []
        node_times_after = []
        search_node_time = 0
        node_found = False
        for i,node_name in enumerate(self.node_names):
            if node_found == False:
                if node_name == search_node_name:
                    node_found = True
                    search_node_time = self.schedule_times[i]
            elif node_found == True:
                #record the name and time to reach of nodes after the node names
                nodes_after.append(self.nodes[i]) #add node to the list
                node_times_after.append(self.schedule_times[i]-search_node_time)

        return node_found,search_node_time,nodes_after,node_times_after

    def get_length(self): #get the length of a schedule (time taken to traverse)
        length = 0
        for edge in self.edges:
```

```
            length = length + edge.travel_time
        return length
```

A1.6 evaluator.py

```
class Evaluator:
    #initalise the evaluators with the standard costs of a system
    def __init__(self,eval_csv,parameters_csv):
        self.vehicle_cost = eval_csv["Vehicle Cost"].to_list()[0] #marginal cost of
            running a vehicle, $/hour
        self.agent_cost_seated = eval_csv["Agent Cost Seated"].to_list()[0] #marginal
            value of agents time, $/seated
        self.agent_cost_standing = eval_csv["Agent Cost Standing"].to_list()[0] #
            marginal value of agents time, higher because standing is unpleasant $/hr
        self.agent_cost_waiting = eval_csv["Agent Cost Waiting"].to_list()[0] #marginal
            value of agents time, higher because waiting is unpleasant $/hr
        self.unfinished_penalty = eval_csv["Unfinished Penalty"].to_list()[0] #penalty
            if passengers are unable to reach their destination, based roughly on cost
            of late night taxi ride
        self.vehicle_max_seated = parameters_csv["Vehicle Max Seated"].to_list()[0] #
            maximum number who can sit inside a vehicle
        self.vehicle_max_standing = parameters_csv["Vehicle Max Standing"].to_list()[0]
            #maximum number who can fit inside a vehicle seated + standing
        self.timesteps_per_hour = 60

    def evaluate(self,sim_times,sim_vehicle_passengers,sim_node_passengers,
        num_failed_passengers,num_successful_passengers):
        seated_passenger_time = 0 #amount of minutes passengers spend seated
        waiting_passenger_time = 0 #amount they spend waiting
        standing_passenger_time = 0 #amount they standing
        vehicle_time = 0 #amount of minutes vehicles are used for
        max_num_vehicles_at_once = 0
        max_passengers_in_a_vehicle = 0
        for i,time in enumerate(sim_times):
            #go through all the time_steps and extract relevant data
            vehicle_passengers = sim_vehicle_passengers[i]
```

```
            node_passengers = sim_node_passengers[i]
            new_seated_time, new_standing_time, num_vehicles, max_passengers = self.passenger_time_vehicles(vehicle_passengers)
            new_waiting_time = self.passenger_time_nodes(node_passengers)
            seated_passenger_time = seated_passenger_time + new_seated_time
            standing_passenger_time = standing_passenger_time + new_standing_time
            waiting_passenger_time = waiting_passenger_time + new_waiting_time
            vehicle_time = vehicle_time + num_vehicles
            if num_vehicles>max_num_vehicles_at_once:
                max_num_vehicles_at_once = num_vehicles
            if max_passengers>max_passengers_in_a_vehicle:
                max_passengers_in_a_vehicle = max_passengers
        #convert resource use time from minutes into hours
        seated_passenger_time = seated_passenger_time/self.timesteps_per_hour #amount of minutes passengers spend seated
        waiting_passenger_time = waiting_passenger_time/self.timesteps_per_hour #amount they spend waiting
        standing_passenger_time = waiting_passenger_time/self.timesteps_per_hour #amount they standing
        vehicle_time = vehicle_time/self.timesteps_per_hour #amount of minutes vehicles are used for
        total_passenger_time = seated_passenger_time + waiting_passenger_time + standing_passenger_time
        cost_seated_passenger_time = standing_passenger_time*self.agent_cost_seated
        cost_standing_passenger_time = standing_passenger_time*self.agent_cost_standing
        cost_waiting_passenger_time = waiting_passenger_time*self.agent_cost_waiting
        cost_passenger_time = cost_seated_passenger_time + cost_standing_passenger_time + cost_waiting_passenger_time
        cost_passenger_failure = num_failed_passengers*self.unfinished_penalty
        cost_vehicle_time = vehicle_time*self.vehicle_cost
        total_cost = cost_passenger_time + cost_vehicle_time + cost_passenger_failure
        #calculate some per capita stats
        num_passengers = num_failed_passengers+num_successful_passengers
        time_per_passenger = (total_passenger_time/num_passengers) #time in hours for each passenger
        time_per_passenger_seated = (seated_passenger_time/num_passengers)
        time_per_passenger_standing = (standing_passenger_time/num_passengers)
```

```
        time_per_passenger_waiting = (waiting_passenger_time/num_passengers)
        failure_rate = (num_failed_passengers/num_passengers)
        cost_per_passenger = cost_vehicle_time/num_passengers#just the financial cost
        total_cost_per_passenger = total_cost/num_passengers #holistic cost
        message = ""
        message = message + "Num Passenger Trips = " + f'{num_passengers:,}' + '\n'
        message = message + "% Trips Did Not Destination = " + f'{(failure_rate*100):.2f}' + '% \n'
        message = message + "Total Time per Passenger = " + f'{(time_per_passenger*self.timesteps_per_hour):.2f}' + ' Mins \n'
        message = message + "Time Standing = " + f'{(time_per_passenger_standing*self.timesteps_per_hour):.2f}' + ' Mins \n'
        message = message + "Time Seated = " + f'{(time_per_passenger_seated*self.timesteps_per_hour):.2f}' + ' Mins \n'
        message = message + "Time Waiting = " + f'{(time_per_passenger_waiting*self.timesteps_per_hour):.2f}' + ' Mins \n'
        message = message + "Cost of Vehicle Operation = $" + f'{cost_vehicle_time:,.0f}' + "\n"
        message = message + "Max Number of Vehicles at Once = " + f'{max_num_vehicles_at_once:,.0f}' + "\n"
        message = message + "Max Passengers in a Vehicle = " + f'{max_passengers_in_a_vehicle:,.0f}' + "\n"
        message = message + "Combined Financial and Time Cost = $" + f'{total_cost:,.2f}' + "\n"
        message = message + "Financial Cost per Passenger = $" + f'{cost_per_passenger:.2f}' + "\n"
        message = message + "Total Cost per Passenger = $" + f'{total_cost_per_passenger:.2f}' + "\n"
        return message

    #get how many minutes passengers were sitting/standing in vehicles at this timestep
    def passenger_time_vehicles(self, vehicle_passengers):
        seated = 0
        standing = 0
        max_passengers = 0
        try:
            num_vehicles = len(vehicle_passengers)
```

```
        except:
            num_vehicles = 0
        for num_passengers in vehicle_passengers:
            if num_passengers>max_passengers:
                max_passengers = num_passengers
            if num_passengers<=self.vehicle_max_seated:
                seated = seated + num_passengers
            else:
                seated = seated + self.vehicle_max_seated
                standing = standing + num_passengers-self.vehicle_max_seated
        return seated,standing,num_vehicles,max_passengers

    #get how many minutes passengers were waiting at nodes at this timestep
    def passenger_time_nodes(self,node_passengers):
        waiting = 0
        for num_passengers in node_passengers:
            waiting = waiting + num_passengers

        return waiting
```

A1.7 render.py

```
#display.py
#responsible for displaying a visualisation of activity on the network

import tkinter as tk
import time as time
import pandas as pd
import numpy as np
import network as n
import evaluator as e
import warnings as warnings
import cProfile as profile
import pstats
from os import path
from pstats import SortKey
```

```


class Display:

    #create the Display object
    def __init__(self):
        self.setup_display_constants() #set display constants which control default appearace of edges and nodes
        self.set_default_flags() #set the flags and modes of the rendering engine to be their default value
        self.setup_window() #create the display window, where all of our GUI will be displayed
        self.setup_canvas() #create the canvas where we can draw edges and nodes and vehicles
        self.setup_main_controls() #setup the widgets which will allow us to control the simulation and visualisation

    ##SETUP FUNCTIONS

    #setup the constants which control the default physical appearance of the network display
    def setup_display_constants(self):
        #node constants
        self.max_node_radius = 30 #maximum node radius if node size scaled
        self.default_node_radius = 5 #node size if nodes unscaled
        self.min_node_radius = 2 #minimum node size if nodes scaled
        self.custom_node_exponent = 3 #how does node radii scale with amount of stuff happening at that node (if nodes scaled)
        self.default_node_colour = 'grey' #node colour if nodes uncoloured
        #edge constants
        self.default_edge_width = 2 #default width of an edge
        self.active_width_addition = 2 #how much will the edge grow in size when clicked on
        self.min_edge_width = 1 #minimum width of an edge if edges scaled
        self.max_edge_width = 10 #maximum width of an edge if edges scaled
        self.custom_edge_exponent = 2 #how does edge width scale with amount of stuff happening at that edge (if edge scaled)
```

```
        self.default_edge_colour = 'black' #what colour will an edge be by default
        self.path_edge_colour = 'magenta' #what colour will an edge which is part of the drawn path be
        self.path_edge_width = 3 #what width will an edge which is part of the drawn path be
        #vehicle constants
        self.default_vehicle_length = 3
        self.default_vehicle_colour = 'blue'
        #node text constants
        self.default_node_text_colour = 'black'
        self.default_edge_text_colour = 'purple'
        #scroll constants
        self.scroll_gain = 1 #how rapid should pan and scanning be
        #id of text above nodes at ends of activated edges, needs to be deleted when the edge is left
        self.text_id_line_end = -1 #default value, to indicate no such object
        self.text_id_line_start = -1 #default value, to indicate no such object
        self.sim_frame_time = 1 #how many seconds between simulation view updates, reciprocal of frame-rate
        #index of vehicle text popups
        self.index_vehicle_text_popup = -1 #default value, to indicate no such object
        self.name_vehicle_text_popup = -1 #default value, to indicate no such object
        #default vehicle capacities, used for determining vehicle colours based on crowding levels
        #note standing capacity is standing + seated capacity
        self.vehicle_seated_capacity = 960 #sydney trains A/B class, 8 carriage
        self.vehicle_standing_capacity = 1680 #sydney trains A/B class, 8 carriage, roughly 4 pax/m^2 open space

    #set the various flags (and modes) used by the rendering engine to their default value
    def set_default_flags(self):
        self.first_render_flag = True #is this the first render of the visualisation for a network?
        self.simulation_setup_flag = False #has the simulation setup (eg trip distribution) been done already?
        self.simulation_run_flag = False #has the simulation been run yet?
```

```
        self.simulation_view_flag = False #is the simulation currently being viewed
        self.simulation_past_vehicles_flag = False #are old vehicles from previous simulations still displayed
        self.secondary_control_mode = 'none' #which set of secondary controls (eg network_viz tools, simulation_viz_tools ) is being displayed
        self.last_node_left_click_index = -1 #index of last node left-clicked, -1 indicates that no nodes have been clicked yet
        self.last_node_right_click_index = -1 #index of last node right-clicked, -1 indicates that no nodes have been right clicked yet
        self.path_edge_arrows = True #will arrows be drawn on plotted routes between nodes, indicating direction of travel

    #setup the window object, in which all of our GUI will be contained
    def setup_window(self):
        window = tk.Tk()
        window.attributes("-fullscreen", True) #make the window full screen
        #window.eval('tk::PlaceWindow . center')
        window.title('Network Simulation')
        window_width = window.winfo_screenwidth()
        window_height = window.winfo_screenheight()
        center_x = int(window_width/2)
        center_y = int(window_height/2)
        #window.geometry(f'{window_width}x{window_height}+{center_x}+{center_y}')
        #window.geometry(f'{window_width}x{window_height}') #this code renders the window in the corret position
        self.window = window

    #setup the canvas object, on which we draw our represention of the network
    def setup_canvas(self):
        window_width = self.window.winfo_screenwidth()
        window_height = self.window.winfo_screenheight()
        self.canvas_width = window_width-440
        self.canvas_height = window_height-100
        self.canvas_center_x = int(self.canvas_width/2)
        self.canvas_center_y = int(self.canvas_height/2)
        self.canvas = tk.Canvas(self.window, bg="white", height=self.canvas_height, width=self.canvas_width)
```

```
        self.canvas.pack(side = tk.RIGHT)
        #bind canvas to scroll options
        self.canvas.bind("<MouseWheel>", self.zoom_canvas)
        self.canvas.bind("<ButtonPress-1>", self.pan_start)
        self.canvas.bind("<B1-Motion>", self.pan_end)
        self.current_zoom = 1 #current zoom level
        self.current_zoom_offset_x = 0 #how much is the display x origin offset from the
            true x origin
        self.current_zoom_offset_y = 0 #how much is the display y origin offset from the
            true y origin

    #setup the main control options
    def setup_main_controls(self):
        #create the control panel
        self.main_controls = tk.Frame(master=self.window)
        #self.main_controls.pack(side = tk.LEFT, anchor=tk.N)
        self.main_controls.place(x=0,y=50)
        #default file paths
        default_nodes = 'nodes_sydney.csv'
        default_edges = 'edges_sydney.csv'
        default_schedule = 'schedule_sydney.csv'
        default_segment_schedule = 'schedule_segments_sydney.csv'
        default_parameters = 'parameters_sydney.csv'
        default_eval = 'eval_sydney.csv'
        default_scenario = 'ScenarioFixed.csv'
        #options
        #verbose option, determines level of logging to the console
        self.verbose = -1 #default level of logging is 0=none, 1=verbose, 2=super
            verbose, -1 is placeholder for setup
        self.verbose_button = tk.Button(master=self.main_controls,fg='black',bg='white',
            command=self.verbose_button_click,width=20)
        self.verbose_button.pack(side = tk.TOP)
        self.verbose_button_click() #display initial message
        #label and input to import node files
        self.node_file_path_label = tk.Label(master=self.main_controls,text='NODE FILE
            PATH',fg='black',bg='white',width=20)
        self.node_file_path_label.pack()
```

```
        self.node_file_path_entry = tk.Entry(master=self.main_controls,fg='black',bg='white',width=20)
        self.node_file_path_entry.insert(0,default_nodes)
        self.node_file_path_entry.pack()
        #label and input to import edge files
        self.edge_file_path_label = tk.Label(master=self.main_controls,text='EDGE FILE PATH',fg='black',bg='white',width=20)
        self.edge_file_path_label.pack()
        self.edge_file_path_entry = tk.Entry(master=self.main_controls,fg='black',bg='white',width=20)
        self.edge_file_path_entry.insert(0,default_edges)
        self.edge_file_path_entry.pack()
        #label and input to import schedule files
        self.schedule_file_path_label = tk.Label(master=self.main_controls,text='SCHEDULE FILE PATH',fg='black',bg='white',width=20)
        self.schedule_file_path_label.pack()
        self.schedule_file_path_entry = tk.Entry(master=self.main_controls,fg='black',bg='white',width=20)
        self.schedule_file_path_entry.insert(0,default_schedule)
        self.schedule_file_path_entry.pack()
        #including the segment files which are used to construct more complex schedules
        self.schedule_segment_file_path_label = tk.Label(master=self.main_controls,text='SEGMENTS FILE PATH',fg='black',bg='white',width=20)
        self.schedule_segment_file_path_label.pack()
        #note that an empty schedule will cause us to use the simple method of schedule extraction
        self.schedule_segment_file_path_entry = tk.Entry(master=self.main_controls,fg='black',bg='white',width=20)
        self.schedule_segment_file_path_entry.insert(0,default_segment_schedule)
        self.schedule_segment_file_path_entry.pack()
        #csv file for importing the network parameters
        self.parameters_file_path_label = tk.Label(master=self.main_controls,text='PARAMETERS FILE PATH',fg='black',bg='white',width=20)
        self.parameters_file_path_label.pack()
        self.parameters_file_path_entry = tk.Entry(master=self.main_controls,fg='black',bg='white',width=20)
        self.parameters_file_path_entry.insert(0,default_parameters)
```

```
        self.parameters_file_path_entry.pack()
        #csv file for importing evaluation costs
        self.eval_file_path_label = tk.Label(master=self.main_controls,text='EVALUATION
            FILE PATH',fg='black',bg='white',width=20)
        self.eval_file_path_label.pack()
        self.eval_file_path_entry = tk.Entry(master=self.main_controls,fg='black',bg='
            white',width=20)
        self.eval_file_path_entry.insert(0,default_eval)
        self.eval_file_path_entry.pack()
        #csv file for importing scenario info
        self.scenario_file_path_label = tk.Label(master=self.main_controls,text='
            SCENARIO FILE PATH',fg='black',bg='white',width=20)
        self.scenario_file_path_label.pack()
        self.scenario_file_path_entry = tk.Entry(master=self.main_controls,fg='black',bg
            ='white',width=20)
        self.scenario_file_path_entry.insert(0,default_scenario)
        self.scenario_file_path_entry.pack()
        #control for importing files
        self.import_files_button = tk.Button(master=self.main_controls,text='IMPORT
            FILES',fg='black',bg='white',command=self.import_files_click,width=20)
        self.import_files_button.pack()
        #this button will draw the network
        self.draw_network_button = tk.Button(master=self.main_controls,text="DRAW
            NETWORK",fg='black',bg='white',command=self.draw_network_click,width=20)
        self.draw_network_button.pack()
        #create a button to select whether to display all node names
        self.node_names_button = tk.Button(master=self.main_controls,text="HOVER NODE
            NAMES",fg='black',bg='white',command=self.node_names_click,width=20,height
            =1)
        self.node_names_button.pack()
        self.node_names_mode = 'no_names'
        #this button will setup the simulation
        self.setup_simulation_button = tk.Button(master=self.main_controls,text="SETUP
            SIMULATION",fg='black',bg='white',command=self.setup_simulation_click,width
            =20)
        self.setup_simulation_button.pack()
        #this button will run the basic simulation
```

```
#self.run_simulation_button = tk.Button(master=self.main_controls, text="RUN SIMULATION",fg='black',bg='white',command=self.run_simulation_click,width=20)
#this option with profiling
self.run_simulation_button = tk.Button(master=self.main_controls,text="RUN SIMULATION",fg='black',bg='white',command=self.run_simulation_click,width=20)
self.run_simulation_button.pack()
#this button will play back the basic simulation
self.view_simulation_button = tk.Button(master=self.main_controls,text="VIEW SIMULATION",fg='black',bg='white',command=self.view_simulation_click,width=20)
self.view_simulation_button.pack()
#this label will provide information to the user
self.message_header = tk.Label(master=self.main_controls,text='MESSAGE',fg='black',bg='white',width=20)
self.message_header.pack()
self.message = tk.Label(master=self.main_controls,text='',fg='black',bg='white',width=20,height=5)
self.message.pack()
#run evaluation button
self.run_evaluation_button = tk.Button(master=self.main_controls,text="RUN EVALUATION",fg='black',bg='white',command=self.run_evaluation_click,width=20)
self.run_evaluation_button.pack()
#set optimiser
self.optimiser_label = tk.Label(master=self.main_controls,text="TIMETABLE OPTIMISER",fg='black',bg='white',width=20)
self.optimiser_label.pack()
self.optimiser_button = tk.Button(master=self.main_controls,text="CSV TIMETABLE",fg='black',bg='white',width=20,command=self.switch_optimiser)
self.optimiser_button.pack()
self.optimiser = 'hardcoded'
#create the underlying visulisation controls
#this button will allow choosing different types of controls
self.secondary_controls = tk.Frame(master=self.window)
self.secondary_controls.place(x=220,y=50)
```

```
        #self.control_mode_select_button = tk.Button(master=self.secondary_controls, text
            ="CONTROL SELECT",fg='black',bg='white',command=self.
            control_mode_select_click, width=20)
        #self.control_mode_select_button.pack()
        #self.control_mode = 'none'
        self.setup_network_viz_tools()
        self.setup_simulation_viz_tools()
        #they are created hidden, and will be unhidden later

    #CLICK FUNCTIONS FOR MAIN CONTROL
    def switch_optimiser(self):
        if self.optimiser=="hardcoded":
            self.optimiser_button.config(text="HENRY CONVEX")
            self.optimiser = 'henry_convex'
        else:
            self.optimiser_button.config(text="CSV TIMETABLE")
            self.optimiser = 'hardcoded'
        if self.simulation_setup_flag==True:
            self.message_update('note you must resetup the simulation to apply a new
                optimiser')

    def run_evaluation_click(self):
        if self.simulation_run_flag==True:
            evaluator_message = self.evaluator.evaluate(self.sim_times,self.
                sim_vehicle_passengers,self.sim_node_passengers,self.
                num_failed_passengers,self.num_successful_passengers)
            self.message_update('please see terminal\n for evaluation printout')
            print(evaluator_message)

        elif self.simulation_run_flag==False:
            self.message_update('simulation must be \n run for evaluation')
            self.log_print('simulation must be run for evaluation')

    #callback for button which allows us to switch between control modes for viewing
        network info vs controls for viewing simulation results
    def control_mode_select_click(self):
        #update control mode
```

```
        if self.control_mode == 'none':
            if self.simulation_setup_flag == True:
                self.control_mode = 'network_viz'
            elif self.simulation_run_flag == True:
                self.control_mode = 'simulation_viz'
            else:
                self.log_print("SETUP AND RUN SIMULATION TO VIEW RESULTS")
                self.message_update("SETUP AND RUN SIMULATION \n TO VIEW RESULTS")
                self.control_mode = 'none'
        elif self.control_mode == 'network_viz':
            if self.simulation_run_flag == True:
                self.control_mode = 'simulation_viz'
            else:
                self.log_print("SETUP AND RUN SIMULATION TO VIEW RESULTS")
                self.message_update("RUN SIMULATION \n TO VIEW SIMULATION RESULTS")
                self.control_mode = 'none'

        elif self.control_mode == 'simulation_viz':
            self.control_mode = 'none'

        self.control_mode_update() #now that the control has been selected, perform the
            tasks associated with updating the control mode

    #update the displayed controls so that we can switch between viewing different types
        of info
    def control_mode_update(self):
        if self.control_mode == 'none':
            #no controls will be displayed
            self.control_mode_select_button.config(text='CONTROL SELECT')
            self.clear_network_viz_tools()
            self.clear_simulation_viz_tools()
        elif self.control_mode == 'network_viz':
            #display controls for viewing unsimulated aspects of the network
            self.control_mode_select_button.config(text='NETWORK VIEW CONTROLS')
            self.clear_simulation_viz_tools()
            self.view_network_viz_tools()
        elif self.control_mode == 'simulation_viz':
```

```
            #display controls for viewing simulation results
            self.control_mode_select_button.config(text='SIMULATION VIEW CONTROLS')
            self.clear_network_viz_tools()
            self.view_simulation_viz_tools()

    #attempt to import the selected files
    def import_files_click(self):
        #extract the file paths from the entry widgets
        node_files_path = self.node_file_path_entry.get()
        edge_files_path = self.edge_file_path_entry.get()
        schedule_files_path = self.schedule_file_path_entry.get()
        schedule_segment_files_path = self.schedule_segment_file_path_entry.get()
        parameter_files_path = self.parameters_file_path_entry.get()
        eval_files_path = self.eval_file_path_entry.get()
        scenario_files_path = self.scenario_file_path_entry.get()
        #check that each file path is valid, and if so, import the file
        node_path_valid = path.isfile(node_files_path)
        edge_path_valid = path.isfile(edge_files_path)
        schedule_path_valid = path.isfile(schedule_files_path)
        parameter_path_valid = path.isfile(parameter_files_path)
        eval_path_valid = path.isfile(eval_files_path)
        scenario_path_valid = path.isfile(scenario_files_path)
        #determine type of schedule
        if schedule_segment_files_path == "":
            #we won't be using schedule segments to construct our schedule
            self.schedule_type = "simple"
            segment_path_valid = True #we are not using the segment path, so it might as
                well be valid
            self.log_print("using simple schedule generation")
        else:
            #we will be using schedule segments to construct our schedule
            self.schedule_type = "complex"
            segment_path_valid = path.isfile(schedule_segment_files_path)
            self.log_print("using complex schedule generation")

        #if user path invalid, inform the user of this
        import_files_message = ""
```

```
        import_successful = True #assume we imported unless it fails
        if node_path_valid==False:
            import_files_message = import_files_message + node_files_path + " is not a
                valid file \n"
            self.log_print(node_files_path + " is not a valid file")
            import_successful = False
        if edge_path_valid==False:
            import_files_message = import_files_message + edge_files_path + " is not a
                valid file \n"
            self.log_print(edge_files_path + " is not a valid file")
            import_successful = False
        if schedule_path_valid==False:
            import_files_message = import_files_message + schedule_files_path + " is not
                 a valid file \n"
            self.log_print(schedule_files_path + " is not a valid file")
            import_successful = False
        if segment_path_valid == False:
            import_files_message = import_files_message + schedule_segment_files_path +
                " is not a valid file \n"
            self.log_print(schedule_segment_files_path + " is not a valid file")
            import_successful = False
        if parameter_path_valid == False:
            import_files_message = import_files_message + parameter_files_path + " is
                not a valid file \n"
            self.log_print(parameter_files_path + " is not a valid file")
            import_successful = False
        if eval_path_valid == False:
            import_files_message = import_files_message + eval_files_path + " is not a
                valid file \n"
            self.log_print(eval_files_path + " is not a valid file")
            import_successful = False
        if scenario_path_valid == False:
            import_files_message = import_files_message + scenario_files_path + " is not
                 a valid file \n"
            self.log_print(scenario_files_path + " is not a valid file")
            import_successful = False

```

```
        if import_successful:
            #if file path is valid, actually import the files
            import_files_message = import_files_message + "files are valid \n"
            #try and import the nodes
            try:
                self.nodes_csv = pd.read_csv(node_files_path,thousands=r',')
            except:
                import_files_message = import_files_message + " import of " +
                    node_files_path + " failed \n not a valid csv file\n"
                import_successful = False
            #try and import the edges
            try:
                self.edges_csv = pd.read_csv(edge_files_path,thousands=r',')
            except:
                import_files_message = import_files_message + " import of " +
                    edge_files_path + " failed  \n not a valid csv file\n"
                import_successful = False
            #try and import the schedule
            try:
                self.schedule_csv = pd.read_csv(schedule_files_path,thousands=r',')
            except:
                import_files_message = import_files_message + " import of " +
                    schedule_files_path + " failed  \n not a valid csv file\n"
                import_successful = False
            #try and import the network/simulation parameters
            try:
                self.parameter_csv = pd.read_csv(parameter_files_path,thousands=r',')
            except:
                import_files_message = import_files_message + " import of " +
                    parameter_files_path  + " failed  \n not a valid csv file\n"
                import_successful = False
            try:
                self.eval_csv = pd.read_csv(eval_files_path,thousands=r',')
            except:
                import_files_message = import_files_message + " import of " +
                    eval_files_path  + " failed  \n not a valid csv file\n"
                import_successful = False
```

```
            try:
                self.scenario_csv = pd.read_csv(scenario_files_path,thousands=r',')
            except:
                import_files_message = import_files_message + " import of " + scenario_files_path + " failed \n not a valid csv file\n"
            #if we are in complex schedule mode, try and import segment info
            if self.schedule_type=='complex':
                try:
                    self.schedule_segments_csv = pd.read_csv(schedule_segment_files_path,thousands=r',',keep_default_na=False)
                    #keep_default_na false so that empty values in a column are
                except:
                    import_files_message = import_files_message + " import of " + schedule_segment_files_path + " failed \n not a valid csv file\n"
                    import_successful = False
            elif self.schedule_type=='simple':
                self.schedule_segments_csv = "" #we don't need the schedule segments file in simple scheduling

        #print a relevant message if import successful
        if import_successful:
            import_files_message = import_files_message + " files imported successfully"
            self.simulation_setup_flag = False #we have not setup the simulation for the new files
        else:
            import_files_message = import_files_message + " file import failed"

        #print the message about the result of importing files
        self.message_update(import_files_message)

    #draw the network from the imported files
    def draw_network_click(self):
        if self.first_render_flag==False:
            self.erase_network_graph()
            self.erase_all_nodes_text('both')
            self.erase_all_edges_text()
```

```
        if self.simulation_setup_flag:
            self.erase_all_edges_text()
        self.extract_nodes_graph()
        self.calculate_node_position()
        self.extract_edges_graph()
        self.calculate_edges_midpoints()
        self.render_graph()
        self.first_render_flag = False

    #setup the simulated network
    def setup_simulation_click(self):
        #we need to draw the network before we can setup up the simulation
        if self.first_render_flag==True:
            self.draw_network_click()

        time1 = time.time()
        self.sim_network = n.Network(nodes_csv=self.nodes_csv, edges_csv=self.edges_csv, schedule_csv=self.schedule_csv, parameters_csv=self.parameter_csv, verbose=self.verbose, segment_csv=self.schedule_segments_csv, eval_csv=self.eval_csv, scenario_csv=self.scenario_csv, schedule_type=self.schedule_type, optimiser=self.optimiser)
        time2 = time.time()
        simulation_setup_message = "simulation setup in \n" + "{:.3f}".format(time2-time1) + " seconds"
        self.log_print(simulation_setup_message)
        self.message_update(simulation_setup_message)
        #if self.simulation_setup_flag:  #only setup network visulisation tools if they have not already been created
        #    #if they have been recreated we need to destroy the old tools
        #    self.clear_network_viz_tools()
        #    self.setup_network_viz_tools()
        #else:
        #    self.setup_network_viz_tools() #setup tools for exploring aspects of the simulated network
        #also setup the evaulator
        self.setup_evaluator()
```

```
        self.simulation_setup_flag = True #flag to indicate that the simulation has been setup

    def setup_evaluator(self):
        self.evaluator = e.Evaluator(self.eval_csv, self.parameter_csv)

    #run the simulation click using Cprofile to determine running times
    def profile_run_simulation_click(self):
        profile.runctx("self.run_simulation_click()",globals(),locals(), 'restats')
        #print how long inside the function call does each called function take
        p = pstats.Stats('restats')
        p.strip_dirs()
        p.sort_stats(SortKey.TIME)
        p.print_stats()

    #run the basic simulation
    def run_simulation_click(self):
        if self.simulation_setup_flag == True:
            simulation_start_message = 'simulation started'
            self.log_print(simulation_start_message)
            self.message_update(simulation_start_message)
            time1 = time.time()
            self.sim_times, self.sim_vehicle_latitudes, self.sim_vehicle_longitudes, self.sim_vehicle_names, self.sim_vehicle_passengers, self.sim_node_passengers, self.num_failed_passengers, self.num_successful_passengers, self.sim_time_taken = self.sim_network.basic_sim() #run the simulation and store the data
            self.setup_default_sim_current_values() #set default values for information about specific timesteps
            self.simulation_run_flag = True #simulation has been run and relevant values have been stored
            time2 = time.time()
            simulation_finished_message = "simulation finished in \n " + "{:.3f}".format(time2-time1) + " seconds \n The simulation represented \n" + str(self.sim_time_taken) + " minutes"
            self.log_print(simulation_finished_message)
            self.message_update(simulation_finished_message)
```

```
        else:
            self.message_update('simulation not yet setup \n cannot run')
            self.log_print('simulation not yet setup cannot run')

    #set default values for current sim variables, to avoid errors if we try and render
        them outside of a timestep
    def setup_default_sim_current_values(self):
        num_nodes = len(self.node_names)
        self.sim_node_current_passengers = np.zeros(num_nodes)
        self.sim_vehicles_current_names = []
        self.sim_vehicles_current_latitudes = []
        self.sim_vehicles_current_longitudes = []
        self.sim_vehicles_current_passengers = []
        self.sim_vehicles_current_colour = []
        self.sim_vehicles_current_length = []

    def view_simulation_click(self):
        if self.simulation_run_flag == False: #simulation needs to be run to be
            displayed
            self.message_update('simulation not yet run \n run simulation to view
                results')
            self.log_print('simulation not yet run,  run simulation to view results')
        elif self.simulation_run_flag == True:
            #go through all time
            if self.simulation_view_flag == True:
                #continue the current simulation if it is already being viewed
                self.message_update('simulation already \n being viewed')
                self.log_print('simulation already being viewed')
            else:
                if self.simulation_past_vehicles_flag == True:
                    #we need to delete any lingering past vehicles
                    self.derender_vehicles(override=True)
                self.num_sim_times = len(self.sim_times)
                time_index = 0
                #self.render_simulation_update(time_index)
                self.render_simulation_update(time_index)

```

```
    #render a simulation update after a delay
    def render_simulation_update(self,index):
        if self.paused == False: #if we are playing back the simulation, play the next frame
            start_render_time = time.time() #get the time at the start of renderings
            end_render_time = start_render_time + self.sim_frame_time #calculate what time we need to move to the next frame to maintain a steady frame-rate
            #extract the data for the current timestep
            sim_time = self.sim_times[index]
            #update the time display
            time_text = 'TIME ' + str(sim_time)
            self.time_label.config(text=time_text)
            #extract other information from the calculate vehicles
            self.extract_current_vehicles_info(index) #extract info about the vehicles in the current simulation timesteps
            self.update_vehicle_text_index() #update the index of the vehicle whose info we are displaying as a popup
            self.extract_current_nodes_info(index) #extract info about the nodes in the current simulation timesteps
            self.calculate_vehicle_position() #calculate the position of the vehicles in the network
            self.simulation_view_flag = True #simulation view has been setup
            self.update_nodes()
            self.update_text_same_node()
            self.generate_edge_overlay_text()
            #after rendering, wait till we reach the time set for the next visual update
            remaining_frame_time = end_render_time-time.time()
            index = index + 1 #index of the next batch of data
            if index>=self.num_sim_times: #we have finished displaying the simulation
                self.log_print("Simulation Display Finished")
                self.message_update("Simulated Display Finished")
                self.simulation_view_flag = False #simulation is no longer being run
                self.simulation_past_vehicles_flag = True #past vehicles still exist that will need to be deleted if we replay the simulation
            elif index < self.num_sim_times:
                #call the callback again once we have waited long enough
```

```python
                self.time_label.after(int(remaining_frame_time*1000),self.render_simulation_update,index)
        if self.paused == True:
            self.time_label.after(10,self.render_simulation_update,index) #check to see if we are still paused 100 times per second

    #update the index of the vehicle whose info we are displaying as a popup
    def update_vehicle_text_index(self):
        try:
            #get the new index of the vehicle
            new_index = self.sim_vehicles_current_names.index(self.name_vehicle_text_popup)
        except ValueError:
            #in this case, the vehicle no longer exists
            #hence delete text popups
            #delete text popups
            self.derender_hover_vehicle_text()
            #and reset index of vehicle whom we are providing info about
            self.name_vehicle_text_popup = -1
            self.index_vehicle_text_popup = -1
        else:
            #change the stored index to reflect the new position in the list of current vehicles
            self.index_vehicle_text_popup = new_index

    def extract_current_vehicles_info(self,index):
        #extract the info for the current time (given by index)
        self.sim_vehicles_current_names = self.sim_vehicle_names[index]
        self.sim_vehicles_current_latitudes = self.sim_vehicle_latitudes[index]
        self.sim_vehicles_current_longitudes = self.sim_vehicle_longitudes[index]
        self.sim_vehicles_current_passengers = self.sim_vehicle_passengers[index]

    def extract_current_nodes_info(self,index):
        #extract the info for the current time (given by index)
        self.sim_node_current_passengers = self.sim_node_passengers[index]

    #switch logging levels (verbosity level)
```

```
    def verbose_button_click(self):
        if self.verbose==0:
            self.verbose_button.config(text='VERBOSE')
            self.verbose = 1
        elif self.verbose==1:
            self.verbose_button.config(text='SUPER VERBOSE')
            self.verbose = 2
        else: #if verbosity already at highest level or is unset, select minimum
            verbosity
            self.verbose_button.config(text='NO LOGGING')
            self.verbose = 0
        if self.simulation_setup_flag == True:#also update the logging level in the
            simulation if it exists
            self.log_print('SIMULATION LOG LEVEL UPDATED TO '+ str(self.verbose),2)
            self.sim_network.verbose = self.verbose

    #NETWORK VIZ TOOLS
    #tools for exploring aspects of the simulated network which do not depend on actual
        simulation
    #eg ideal journey times and paths, and passenger trip distribution

    #setup these tools
    def setup_network_viz_tools(self):
        #create the overall frame
        self.network_viz = tk.Frame(master=self.secondary_controls)
        #self.network_viz.pack(side = tk.TOP)
        self.network_viz_label = tk.Label(master=self.network_viz, text='NETWORK VIEW
            OPTIONS', fg='white', bg='cyan', width=20)
        self.network_viz_label.pack()
        #A label for the too/from select button
        self.display_mode_label = tk.Label(master=self.network_viz, text='DIRECTION
            SELECT', fg='black', bg='white', width=20)
        self.display_mode_label.pack()
        #create a button to choose whether we are viewing information "from" a node or "
            too" a node
        self.too_from_select_button = tk.Button(master=self.network_viz, text="FROM NODE"
            , fg='black', bg='white', command=self.too_from_select_click, width=20)
```

```
            self.too_from_select_button.pack()
            self.from_node = True #True = from_node, False= too_node
            #create a button to choose which edge direction will be used for edge related plotting
            self.edge_direction_button = tk.Button(master=self.network_viz,text="BOTH EDGE DIRECTIONS",fg='black',bg='white',command=self.edge_direction_select_click, width=20)
            self.edge_direction_button.pack()
            self.edge_direction_mode = 'both'
            #A label for the nodes numeric overlay button
            self.nodes_numeric_overlay_label = tk.Label(master=self.network_viz,text='NUMERIC OVERLAY MODE',fg='black',bg='white',width=20)
            self.nodes_numeric_overlay_label.pack()
            #create a button to select whether to provide a numeric overlay on the canvas to provide information about node relationships
            self.nodes_numeric_overlay_button = tk.Button(master=self.network_viz,text="NO NODE OVERLAY",fg='black',bg='white',command=self.nodes_numeric_overlay_click,width=20,height=2)
            self.nodes_numeric_overlay_button.pack()
            self.nodes_numeric_overlay_mode = 'no_info'
            #create a button to select whether to provide a numeric overlay on the canvas to provide information about edges
            self.edges_numeric_overlay_button = tk.Button(master=self.network_viz,text="NO EDGE OVERLAY",fg='black',bg='white',command=self.edges_numeric_overlay_click,width=20,height=2)
            self.edges_numeric_overlay_button.pack()
            self.edges_numeric_overlay_mode = 'no_info'
            #A label for the node appearance controls
            self.nodes_appearance_label = tk.Label(master=self.network_viz,text='NODE APPEARANCE',fg='black',bg='white',width=20)
            self.nodes_appearance_label.pack()
            #create a button to select whether to use the size of nodes to provide information about nodes and their relationships
            self.node_size_button = tk.Button(master=self.network_viz,text="CONSTANT NODE SIZE",fg='black',bg='white',command=self.node_size_click,width=20,height=2)
            self.node_size_button.pack()
            self.node_size_type = "constant" #by default, nodes will be a constant size
```

```
        #a button to select whether to use the colour of nodes to provide information about node relationships
        self.node_colour_button = tk.Button(master=self.network_viz, text="CONSTANT NODE COLOUR", fg='black', bg='white', command=self.node_colour_click, width=20, height=2)
        self.node_colour_button.pack()
        self.node_colour_type = "constant"
        #A label for the edge appearance controls
        self.edges_appearance_label = tk.Label(master=self.network_viz, text='EDGE APPEARANCE', fg='black', bg='white', width=20)
        self.edges_appearance_label.pack()
        #create a button to select whether to use the size of edges to provide information about the edges
        self.edge_width_button = tk.Button(master=self.network_viz, text="CONSTANT EDGE WIDTH", fg='black', bg='white', command=self.edge_width_click, width=20, height=2)
        self.edge_width_button.pack()
        self.edge_width_type = "constant" #by default, edges will be a constant size
        #create a button to select whether to use the colour of edges to provide information about the edges
        self.edge_colour_button = tk.Button(master=self.network_viz, text="CONSTANT EDGE COLOUR", fg='black', bg='white', command=self.edge_colour_click, width=20, height=2)
        self.edge_colour_button.pack()
        self.edge_colour_type = "constant" #by default, edges will be a constant size
        self.secondary_control_mode = 'network_viz' #network viz mode is being displayed
        self.network_viz.pack()

    def setup_simulation_viz_tools(self):
        #create the overall frame
        self.secondary_control_mode = 'simulation_viz' #simulation viz mode is being displayed
        self.simulation_viz = tk.Frame(master=self.secondary_controls)
        #label for simulation viz mode
        self.simulation_viz_label = tk.Label(master=self.simulation_viz, text='SIM VIEW OPTIONS', fg='white', bg='cyan', width=20)
        self.simulation_viz_label.pack()
```

```
        #create a label to display the time
        self.time_label = tk.Label(master=self.simulation_viz,text='TIME',fg='black',bg='white',width=20)
        self.time_label.pack()
        #create a button to enable us to control whether the simulation is running
        self.pause_play_button = tk.Button(master=self.simulation_viz,text='PLAYING',fg='black',bg='white',width=20,command=self.pause_play_button_click)
        self.paused = False #simulation visualisation starts paused
        self.pause_play_button.pack()
        #create controlsn to enable us to control the speed of the simulation
        #first a label to indicate this
        #note the label is set for self.sim_frame_time = 1
        self.simulation_speed_label = tk.Label(master=self.simulation_viz,text='UPDATES PER SECOND = 1',fg='black',bg='white',width=20)
        self.simulation_speed_label.pack()
        #add a entry to enable us to enter the frame speed
        self.simulation_speed_entry = tk.Entry(master=self.simulation_viz,fg='black',bg='white',width=20)
        self.simulation_speed_entry.insert(0,self.sim_frame_time)
        self.simulation_speed_entry.pack()
        #add a button to update the frame speed
        self.simulation_speed_update_button = tk.Button(master=self.simulation_viz,text='UPDATE SPEED',fg='black',bg='white',command=self.simulation_speed_update_click,width=20)
        self.simulation_speed_update_button.pack()
        #add controls for vehicle appearance rendering
        self.vehicle_appearance_label = tk.Label(master=self.simulation_viz,text='VEHICLE APPEARANCE',fg='black',bg='white',width=20)
        self.vehicle_appearance_label.pack()
        #add button to control vehicle colour
        self.vehicle_colour_button = tk.Button(master=self.simulation_viz,text="VEHICLE COLOUR BASED \n ON CROWDING",fg='black',bg='white',command=self.vehicle_colour_click,width=20,height=2)
        self.vehicle_colour_type = "crowding" #by default, vehicle colours will be based off the level of crowding in the vehicle
        self.vehicle_colour_button.pack()
        self.simulation_viz.pack()
```

```
    def vehicle_colour_click(self):
        if self.vehicle_colour_type == "crowding":
            self.vehicle_colour_type = "constant"
        elif self.vehicle_colour_type == "constant":
            self.vehicle_colour_type = "crowding"

        self.vehicle_colour_button_text_update()
        self.calculate_vehicle_position #rerender vehicles to match the new colour
            scheme

    def vehicle_colour_button_text_update(self):
        if self.vehicle_colour_type == "crowding":
            self.vehicle_colour_button.config(text="VEHICLE COLOUR BASED \n ON CROWDING"
                )
        elif self.vehicle_colour_type == "constant":
            self.vehicle_colour_button.config(text="CONSTANT")


    def simulation_speed_update_click(self):
        #extract the new updates per second
        new_updates_per_second = self.simulation_speed_entry.get()
        try:
            #if it can be convert to a float
            new_updates_per_second = float(new_updates_per_second)

        except:
            error_text = str(new_updates_per_second) + " Is not numeric, please enter a
                numeric frame-rate"
            self.log_print(error_text)
        else:
            #calculate the new time between frames
            self.sim_frame_time = 1/new_updates_per_second
            updates_per_second_text = 'UPDATES/SECOND = ' + str(new_updates_per_second)
            self.simulation_speed_label.config(text=updates_per_second_text)
```

```
    #control whether the simulation visulisation is paused or playing
    def pause_play_button_click(self):
        if self.paused == True:
            self.paused = False
            self.pause_play_button.config(text="PLAYING")
        elif self.paused == False:
            self.paused = True
            self.pause_play_button.config(text="PAUSED")

    #hide the network_viz tool controls
    def clear_network_viz_tools(self):
        if self.secondary_control_mode == 'network_viz':
            self.network_viz.pack_forget() #hide the network viz controls

    #redisplay the network_viz tools
    def view_network_viz_tools(self):
        self.network_viz.pack(side = tk.TOP)
        self.secondary_control_mode = 'network_viz'

    #hide the network_viz tool controls
    def clear_simulation_viz_tools(self):
        if self.secondary_control_mode == 'simulation_viz':
            self.simulation_viz.pack_forget() #hide the simulation viz controls

    def view_simulation_viz_tools(self):
        self.simulation_viz.pack(side = tk.TOP)
        self.secondary_control_mode = 'simulation_viz'

    #CLICK FUNCTIONS FOR NETWORK VIZ TOOLS
    #command for button to switch between displaying and not displaying node names
    def node_names_click(self):
        #update mode
        if self.node_names_mode == 'no_names':
            self.node_names_mode = 'display_names'
        elif self.node_names_mode == 'display_names':
            self.node_names_mode = 'no_names'
        #perform the actual update of the button and the rendering
```

```
        self.node_names_update()

    #perform the actual update between displaying and not displaying node names
    def node_names_update(self):
        if self.node_names_mode == 'no_names':
            self.node_names_button.config(text="HOVER NODE NAMES")
            self.erase_all_nodes_text(mode='above') #clear away node name text
        elif self.node_names_mode == 'display_names':
            self.node_names_button.config(text="DISPLAY NODE NAMES")
            self.display_text_info_node(self.node_names,where_mode='above') #display node names on the map


    #command for button to switch whether numeric information (eg num passengers) will be displayed next to all relevant nodes
    def nodes_numeric_overlay_click(self):
        if self.nodes_numeric_overlay_mode == 'no_info': #switch to node total mode, where the total traffic too/from each node is displayed
            self.nodes_numeric_overlay_mode = 'node_total'

        elif self.nodes_numeric_overlay_mode == 'node_total':#switch to node relative mode, where the traffic too/from the key node is displayed
            self.nodes_numeric_overlay_mode = 'node_relative'

        elif self.nodes_numeric_overlay_mode == 'node_relative':#switch to distance mode, where the distance too/from the key node is displayed
            self.nodes_numeric_overlay_mode = 'node_distance'

        elif self.nodes_numeric_overlay_mode == 'node_distance' and self.simulation_run_flag==True:
            #if simulation has been run, switch to a mode where we display the actual number of waiting passengers (at each simulation timestep)
            self.nodes_numeric_overlay_mode = 'waiting_passengers'

        else: #switch back to the default mode of no numeric overlay
            self.nodes_numeric_overlay_mode = 'no_info'

```

```
        self.nodes_numeric_overlay_button_text_update() #update the text on the button
        self.update_text_same_node() #update the numeric overlay

    #update the text in the nodes numeric overlay button
    def nodes_numeric_overlay_button_text_update(self):
        text = "INVALID MODE FOR \n NODES NUMERIC OVERLAY"
        if self.nodes_numeric_overlay_mode == 'no_info':
            text = "NO NODE OVERLAY"
        elif self.nodes_numeric_overlay_mode == 'node_total':
            if self.from_node:
               text="NODES OVERLAY \n TOTAL TRAFFIC FROM NODES"
            else:
                text="NODES OVERLAY \n TOTAL TRAFFIC TOO NODES"
        elif self.nodes_numeric_overlay_mode == 'node_relative':
            if self.from_node:
               text="NODES OVERLAY TRAFFIC \n FROM CLICKED NODE"
            else:
                text="NODES OVERLAY TRAFFIC \n TOO CLICKED NODE"
        elif self.nodes_numeric_overlay_mode == 'node_distance':
            if self.from_node:
               text="NODES OVERLAY DISTANCE \n FROM CLICKED NODE"
            else:
                text="NODES OVERLAY DISTANCE \n TOO CLICKED NODE"

        elif self.nodes_numeric_overlay_mode == 'waiting_passengers':
            text = "NODE OVERLAY \n PASSENGERS AT NODE"
        self.nodes_numeric_overlay_button.config(text=text)

    #command for button to switch whether edge statistics will be displayed forward/reverse
    def edge_direction_select_click(self):
        if self.edge_direction_mode == 'both':
             self.edge_direction_button.config(text='FORWARD EDGE DIRECTION')
             self.edge_direction_mode = 'forward'
        elif self.edge_direction_mode == 'forward':
            self.edge_direction_button.config(text='REVERSE EDGE DIRECTION')
            self.edge_direction_mode = 'reverse'
```

```
        elif self.edge_direction_mode == 'reverse':
            self.edge_direction_button.config(text='BOTH EDGE DIRECTIONS')
            self.edge_direction_mode = 'both'

        self.edges_overlay_button_text_update()  #update the text on the button
        self.generate_edge_overlay_text()        #update the overlay rendering
        self.edge_width_button_text_update() #update the text on the buttons
        self.edge_colour_button_text_update()
        self.update_edges() #update the rendering of the edges

    #command for button to switch whether numeric information (eg num passengers) will be displayed along relevant edges
    def edges_numeric_overlay_click(self):
        if self.edges_numeric_overlay_mode == 'no_info': #switch to distance mode, where the length of the node forward/reverse is displayed
            self.edges_numeric_overlay_mode = 'distance'
        elif self.edges_numeric_overlay_mode == 'distance':
            self.edges_numeric_overlay_mode = 'traffic' #switch to each traffic, where the amount of traffic forward/reverse an edge is displayed
        elif self.edges_numeric_overlay_mode == 'traffic':
            self.edges_numeric_overlay_mode = 'total_traffic' #switch to total traffic, where the combined amount of traffic on an edge is displayed
        else:
            self.edges_numeric_overlay_mode = 'no_info' #switch back to the default mode of no numeric overlay


        self.edges_overlay_button_text_update() #update the text on the button
        self.generate_edge_overlay_text()        #update the overlay rendering

    #function to correctly set the text for the edges numeric overlay button
    def edges_overlay_button_text_update(self):
        if self.edges_numeric_overlay_mode == 'no_info':
            self.edges_numeric_overlay_button.config(text="NO EDGE OVERLAY")
        elif self.edges_numeric_overlay_mode == 'distance':
            if self.edge_direction_mode == 'both':
```

```
                self.edges_numeric_overlay_button.config(text="FORWARD + REVERSE \n EDGE
                    TRAVEL TIME")
            elif self.edge_direction_mode == 'forward':
                self.edges_numeric_overlay_button.config(text="FORWARD EDGE \n TRAVEL
                    TIME")
            elif self.edge_direction_mode == 'reverse':
                self.edges_numeric_overlay_button.config(text="REVERSE EDGE \n TRAVEL
                    TIME")

        elif self.edges_numeric_overlay_mode == 'traffic':
            if self.edge_direction_mode == 'both':
                self.edges_numeric_overlay_button.config(text="FORWARD + REVERSE \n EDGE
                    TRAFFIC")
            elif self.edge_direction_mode == 'forward':
                self.edges_numeric_overlay_button.config(text="FORWARD TRAFFIC \n
                    THROUGH EDGE")
            elif self.edge_direction_mode == 'reverse':
                self.edges_numeric_overlay_button.config(text="REVERSE TRAFFIC \n
                    THROUGH EDGE")

        elif self.edges_numeric_overlay_mode == 'total_traffic':
            self.edges_numeric_overlay_button.config(text="TOTAL TRAFFIC \n THROUGH EDGE
                ")

    #command for button to switch between options for setting node size
    def node_size_click(self):
        #switch to the new mode
        if self.node_size_type == "constant":
            #switch to mode where node size is based on traffic going too/from the
                clicked node to other nodes
            self.node_size_type = "node_relative"
        elif self.node_size_type == "node_relative":
            #switch to mode where node size is based on total traffic coming too/from
                the clicked node
            self.node_size_type = "node_total"
        elif self.node_size_type == "node_total":
            #switch to mode where node size is based on the number of waiting passengers
```

```
            self.node_size_type = "node_passengers"
        elif self.node_size_type == "node_passengers":
            self.node_size_type = "constant"
        #update the text of the button
        self.node_size_button_text_update()
        #rerender the nodes to be of the correct size
        self.update_nodes()

    #command for the node size button to update to the correct text for it's mode of
        operation
    def node_size_button_text_update(self):
        if self.node_size_type == "node_relative":
            if self.from_node:
                self.node_size_button.config(text="NODE SIZE TRAFFIC \n FROM CLICKED
                    NODE")
            else:
                self.node_size_button.config(text="NODE SIZE TRAFFIC \n TO CLICKED NODE"
                    )
        elif self.node_size_type == "node_total":
            if self.from_node:
                self.node_size_button.config(text="NODE SIZE TOTAL TRAFFIC \n FROM NODE"
                    )
            else:
                self.node_size_button.config(text="NODE SIZE TOTAL TRAFFIC \n TO NODE")
        elif self.node_size_type == "constant":
            self.node_size_button.config(text="CONSTANT NODE SIZE")
        elif self.node_size_type == "node_passengers":
            self.node_size_button.config(text="NODE SIZE NUM PASSENGES \n WAITING AT
                NODE")

    #command for button to switch between options for setting node colour
    def node_colour_click(self):
        if self.node_colour_type == "constant":
            #switch to mode where node colour is based on journey distance to/from
                clicked node
            self.node_colour_type = "distance"
        elif self.node_colour_type == "distance":
```

```
            #switch to mode where node colour is based on total traffic coming too/from
                the clicked node
            self.node_colour_type = "node_relative"
        elif self.node_colour_type == "node_relative":
            #switch to mode where node colour is based on traffic going too/from the
                clicked node to other nodes
            self.node_colour_type = "node_total"
        elif self.node_colour_type=="node_total":
            #switch to node colour being based on number of passengers waiting at the
                node
            self.node_colour_type = "node_passengers"
        elif self.node_colour_type == "node_passengers":
            #switch to constant node colour
            self.node_colour_type = "constant"


        #update the text of the button
        self.node_colour_button_text_update()
        #rerender the nodes to be of the correct colour
        self.update_nodes()

    #command for the node colour button to update to the correct text for it's mode of
        operation
    def node_colour_button_text_update(self):
        if self.node_colour_type == "distance":
            if self.from_node:
                self.node_colour_button.config(text="NODE COLOUR DISTANCE \n FROM
                    CLICKED NODE")
            else:
                self.node_colour_button.config(text="NODE COLOUR DISTANCE \n TO CLICKED
                    NODE")
        elif self.node_colour_type == "node_relative":
            if self.from_node:
                self.node_colour_button.config(text="NODE COLOUR TRAFFIC \n FROM CLICKED
                    NODE")
            else:
```

```
                self.node_colour_button.config(text="NODE COLOUR TRAFFIC \n TO CLICKED NODE")
        elif self.node_colour_type == "node_total":
            if self.from_node:
                self.node_colour_button.config(text="NODE COLOUR TOTAL TRAFFIC \n FROM NODE")
            else:
                self.node_colour_button.config(text="NODE COLOUR TOTAL TRAFFIC \n TO NODE")
        elif self.node_colour_type=="constant":
            self.node_colour_button.config(text="CONSTANT NODE COLOUR")
        elif self.node_colour_type=="node_passengers":
            self.node_colour_button.config(text="NODE COLOUR NUM PASSENGERS \n WAITING AT NODE")

    #command for the edge width button to switch between options for setting edge width
    def edge_width_click(self):
        #switch to the new mode
        if self.edge_width_type == "constant":
            #switch to mode where edge width is based on traffic going forward/reverse through nodes
            self.edge_width_type = "traffic"
        elif self.edge_width_type == "traffic":
            #switch to mode where edge width is constant
            self.edge_width_type = "constant"

        #update the text of the button
        self.edge_width_button_text_update()
        #rerender the nodes to be of the correct size
        self.update_edges()

    #command for the edge width button to update to the correct text for it's mode of operation
    def edge_width_button_text_update(self):
        if self.edge_width_type == "constant":
            self.edge_width_button.config(text="CONSTANT EDGE WIDTH")
        elif self.edge_width_type == "traffic":
```

```
            if self.edge_direction_mode == 'forward':
                self.edge_width_button.config(text="EDGE WIDTH \n FORWARD TRAFFIC")
            elif self.edge_direction_mode == 'reverse':
                self.edge_width_button.config(text="EDGE WIDTH \n REVERSE TRAFFIC")
            elif self.edge_direction_mode == 'both':
                self.edge_width_button.config(text="EDGE WIDTH \n COMBINED TRAFFIC")

    #command for the edge colour button
    def edge_colour_click(self):
        #switch to the new mode
        if self.edge_colour_type == "constant":
            #switch to mode where edge colour is based on traffic going forward/reverse
                through nodes
            self.edge_colour_type = "traffic"
        elif self.edge_colour_type == "traffic":
            #switch to mode where edge colour is based on travel time
            self.edge_colour_type = "time"
        elif self.edge_colour_type == "time":
            #switch to mode where edge colour is constant
            self.edge_colour_type = "constant"

        #update the text of the button
        self.edge_colour_button_text_update()
        #rerender the nodes to be of the correct size
        self.update_edges()

    #function to update the text of the edge colour button
    def edge_colour_button_text_update(self):
        if self.edge_colour_type == "constant":
            self.edge_colour_button.config(text="CONSTANT EDGE COLOUR")
        elif self.edge_colour_type == "traffic":
            if self.edge_direction_mode == 'forward':
                self.edge_colour_button.config(text="EDGE COLOUR \n FORWARD TRAFFIC")
            elif self.edge_direction_mode == 'reverse':
                self.edge_colour_button.config(text="EDGE COLOUR \n REVERSE TRAFFIC")
            elif self.edge_direction_mode == 'both':
                self.edge_colour_button.config(text="EDGE COLOUR \n COMBINED TRAFFIC")
```

```python

        elif self.edge_colour_type == "time":
            if self.edge_direction_mode == 'forward':
                self.edge_colour_button.config(text="EDGE COLOUR \n FORWARD TRAVEL TIME"
                    )
            elif self.edge_direction_mode == 'reverse':
                self.edge_colour_button.config(text="EDGE COLOUR \n REVERSE TRAVEL TIME"
                    )
            elif self.edge_direction_mode == 'both':
                self.edge_colour_button.config(text="EDGE COLOUR \n AVERAGE TRAVEL TIME"
                    )


    #switch between viewing information too a node or from a node
    def too_from_select_click(self):
        if self.from_node:
            self.from_node = False
            self.too_from_select_button.config(text='TOO NODE')
            self.update_text_same_node()
        else:
            self.from_node = True
            self.too_from_select_button.config(text='FROM NODE')
            self.update_text_same_node()

        #update the other buttons text
        self.nodes_numeric_overlay_button_text_update()
        self.node_colour_button_text_update()
        self.node_size_button_text_update()

    #FUNCTIONS TO DETERMINE NODE SIZE/COLOUR

    #set node sizes in accordance with the mode choosen
    def set_node_sizes(self):
        num_nodes = len(self.node_names)
        if self.node_size_type =="constant":
            self.nodes_radii = [self.default_node_radius]*num_nodes

```

```
        elif self.node_size_type == "node_relative":
            if self.last_node_left_click_index == -1:
                self.nodes_radii = [self.default_node_radius]*num_nodes
                self.message_update("click on a node to set node sizes based on traffic too/from that node") #node sizes will not be updated till users click on a node
            else:
                if self.from_node:
                    trips = self.sim_network.origin_destination_trips[self.last_node_left_click_index,:] #extract number of trips starting from this node
                    total = np.sum(trips)
                else:
                    trips = self.sim_network.origin_destination_trips[:,self.last_node_left_click_index] #extract number of trips going to this node
                    total = np.sum(trips)
                self.calculate_node_sizes(trips,total)

        elif self.node_size_type == "node_total":
            total = np.sum(self.sim_network.origin_destination_trips) #use the total number of trips
            if self.from_node:
                trips = np.sum(self.sim_network.origin_destination_trips,0) #extract number of trips starting from all nodes
            else:
                trips = np.sum(self.sim_network.origin_destination_trips,1) #extract number of trips ending at all nodes
            self.calculate_node_sizes(trips,total)

        elif self.node_size_type == "node_passengers":
            passengers = self.sim_node_current_passengers
            total = np.sum(self.sim_network.origin_destination_trips) #use the total number of trips, we need a constant total to prevent nodes shrinking as number of passengers grows
            self.calculate_node_sizes(passengers,total)
        else:
```

```
            warnings.warn("node_size_type " + self.node_size_type + " not yet impleneted using constant node size instead")
            #other modes not yet implemented, use constant node sizes instead
            self.nodes_radii = [self.default_node_radius]*num_nodes


    #set node colours in accordance with the mode choosen
    def set_node_colours(self):
        num_nodes = len(self.node_names)
        if self.node_colour_type =="constant":
            self.nodes_colour = [self.default_node_colour]*num_nodes

        #set colour based on number of journeys too/from node to clicked node
        elif self.node_colour_type == "node_relative":
            if self.last_node_left_click_index == -1:
                self.nodes_colour = [self.default_node_colour]*num_nodes
                self.message_update("click on a node to set node colours based on traffic too/from that node") #users need to select a node to update the colours
            else:
                if self.from_node:
                    trips = self.sim_network.origin_destination_trips[self.last_node_left_click_index,:] #extract number of trips starting from this node
                    total = np.sum(trips)
                else:
                    trips = self.sim_network.origin_destination_trips[:,self.last_node_left_click_index] #extract number of trips going to this node
                    total = np.sum(trips)
                self.calculate_node_colours(trips,total)

        #set colour based on journeys too/from node in total
        elif self.node_colour_type == "node_total":
            total = np.sum(self.sim_network.origin_destination_trips) #use the total number of trips
            if self.from_node:
```

```
                trips = np.sum(self.sim_network.origin_destination_trips,0) #extract number of trips starting from all nodes
            else:
                trips = np.sum(self.sim_network.origin_destination_trips,1) #extract number of trips ending at all nodes
            self.calculate_node_colours(trips,total)

        #set node colour based on ideal distance to clicked node
        elif self.node_colour_type == "distance":
            if self.last_node_left_click_index == -1:
                self.nodes_colour = [self.default_node_colour]*num_nodes
                self.message_update("click on a node to set node colours based on distance too/from that node") #users need to select a node to update the colours
            else:
                max_time = np.amax(self.sim_network.distance_to_all)
                if self.from_node:
                    times = self.sim_network.distance_to_all[self.last_node_left_click_index,:] #extract journey times starting at this node
                else:
                    times = self.sim_network.distance_to_all[:,self.last_node_left_click_index] #extract journey times going to this node
                self.calculate_node_colours(times,max_time,mode='linear')

        #set node colour based on number of passengers waiting at the node
        elif self.node_colour_type== "node_passengers":
            passengers = self.sim_node_current_passengers
            total = np.sum(self.sim_network.origin_destination_trips) #use the total number of trips, we need a constant total to prevent nodes shrinking as number of passengers grows
            self.calculate_node_colours(passengers,total)

    #calculate_node_sizes based on provided information
    def calculate_node_sizes(self,nodes_quantity,total_quantity,mode='default'):
        total_quantity = total_quantity + 1 #add 1 to prevent divide by zero errors
```

```python
        num_nodes = len(nodes_quantity)
        for i in range(num_nodes):
            node_fraction = nodes_quantity[i]/total_quantity#fraction of total amount
                occuring at that node
            self.nodes_radii[i] = (node_fraction**(1/self.custom_node_exponent))*self.
                max_node_radius
            if self.nodes_radii[i] < self.min_node_radius: #enforce the minimum size of
                a node
                self.nodes_radii[i] = self.min_node_radius

    #calculate and perform final setting of node colour based on provided information
    def calculate_node_colours(self,nodes_quantity,total_quantity,mode='default'):
        num_nodes = len(nodes_quantity)
        for i in range(num_nodes):
            node_fraction = nodes_quantity[i]/total_quantity#fraction of total amount
                occuring at that node
            #determine how far along the spectrum from blue to red through green the
                colour is
            if mode=='default': #use custom scaling (by default cubic), good for
                passenger volumes
                node_colour_fraction = (node_fraction**(1/self.custom_node_exponent))
            elif mode=='linear': #use linear scaling, good for distance to travel in
                smaller maps
                node_colour_fraction = node_fraction
            #convert node_colour_fraction to RGB, blue at 0, green at 0.3, red at 1

            midpoint = 0.3 #midpoint of colour scale is green
            if node_colour_fraction<=midpoint:
                #colour scale is from blue to green
                green = node_colour_fraction/midpoint
                blue =  1-green
                red = 0
            elif node_colour_fraction>midpoint:
                #colour scale is from green to red
                red = (node_colour_fraction-midpoint)/(1-midpoint)
                green = 1-red
                blue = 0
```

```

            #convert 24 bit RGB colour to the hex format expected by tkinter
            self.nodes_colour[i] = RGB_TO_TK_HEX(int(red*255),int(green*255),int(blue
                *255))

    #calculate vehicle colours based on how crowded the vehicles are
    #note at the moment, this code requires all vehicles to all have the same capacity
    def calculate_vehicle_colours_crowding(self,vehicle_num_passengers,seated_capacity,
        standing_capacity):
        #blue is empty, green is half seated capacity, yellow is full seated capacity,
            red is full standing capacity
        num_vehicles = len(vehicle_num_passengers)
        for i in range(num_vehicles):
            this_vehicle_num_passengers = vehicle_num_passengers[i]
            if this_vehicle_num_passengers<=seated_capacity:
                fraction_seated_capacity = this_vehicle_num_passengers/seated_capacity
                midpoint = 0.5
                if fraction_seated_capacity<=midpoint:
                    #for less than half seats occupied
                    #no red, smooth transition from blue to green
                    red = 0
                    green = fraction_seated_capacity/midpoint
                    blue = 1 - green
                else:
                    #for more than half seats occupied but no standing
                    #smooth transition from green to yellow
                    red = (fraction_seated_capacity-midpoint)/(1-midpoint)
                    green = 1
                    blue = 0
            elif this_vehicle_num_passengers<=standing_capacity:
                fraction_standing_capacity = (this_vehicle_num_passengers-
                    seated_capacity)/(standing_capacity-seated_capacity)
                #smooth transition from yellow at no-standing to red at max standing
                red = 1
                green = 1-fraction_standing_capacity
                blue = 0
            else:
```

```
                #for overloaded vehicles
                red = 1
                green = 0
                blue = 0
            #now set the colour of the vehicle
            self.sim_vehicles_current_colour[i] = RGB_TO_TK_HEX(int(red*255),int(green*255),int(blue*255))

    #FUNCTIONS TO DETERINE EDGE WIDTH/COLOUR
    #set edge width based on data about the edge (which data depends on mode)
    def set_edge_widths(self):
        num_edges = len(self.edge_end_indices)
        if self.edge_width_type == "constant":
            self.edge_widths = [self.default_edge_width]*num_edges
        else:
            (forward_edge_data,reverse_edge_data) = self.extract_data_edges(self.edge_width_type)
            if self.edge_direction_mode == 'forward':
                data = np.asarray(forward_edge_data)
            elif self.edge_direction_mode == 'reverse':
                data = np.asarray(reverse_edge_data)
            elif self.edge_direction_mode == 'both':
                if self.edge_width_type == 'traffic':
                    data = np.asarray(reverse_edge_data) + np.asarray(reverse_edge_data) #for both in traffic mode, combine the forward and reverse traffic for display
                elif self.edge_width_type == 'time':
                    data = (np.asarray(reverse_edge_data) + np.asarray(reverse_edge_data))/2 #in time mode(which is not yet implemented) , take the average

            self.calculate_edge_widths(data)

    #calculate and perform final setting of edge width based on provided information
    def calculate_edge_widths(self,edges_quantity,mode='default'):
        num_edges = len(edges_quantity)
        total_quantity = np.max(edges_quantity)
```

```
        for i in range(num_edges):
            edge_fraction = edges_quantity[i]/total_quantity#fraction of total amount
                occuring at that node
            self.edge_widths[i] = (edge_fraction**(1/self.custom_edge_exponent))*self.
                max_edge_width
            if self.edge_widths[i] < self.min_edge_width: #enforce the minimum size of a
                 node
                self.edge_widths[i] = self.min_edge_width

    def set_edge_colours(self):
        num_edges = len(self.edge_end_indices)
        if self.edge_colour_type =="constant":
            self.edge_colours = [self.default_edge_colour]*num_edges
        else:
            (forward_edge_data, reverse_edge_data) = self.extract_data_edges(self.
                edge_colour_type)
            if self.edge_direction_mode == 'forward':
                data = np.asarray(forward_edge_data)
            elif self.edge_direction_mode == 'reverse':
                data = np.asarray(reverse_edge_data)
            elif self.edge_direction_mode == 'both':
                if self.edge_colour_type == 'traffic':
                    data = np.asarray(reverse_edge_data) + np.asarray(reverse_edge_data)
                         #for both in traffic mode, combine the forward and reverse
                        traffic for display
                elif self.edge_colour_type == 'time':
                    data = (np.asarray(reverse_edge_data) + np.asarray(reverse_edge_data
                        ))/2 #in time mode, take the average

            self.calculate_edge_colours(data)

    #calculate and perform final setting of edge width based on provided information
    def calculate_edge_colours(self,edges_quantity,mode='default'):
        num_edges = len(edges_quantity)
        total_quantity = np.max(edges_quantity)
        for i in range(num_edges):
```

```
            edge_fraction = edges_quantity[i]/total_quantity#fraction of total amount occuring at that node
            #determine how far along the spectrum from blue to red through green the colour is
            if mode=='default': #use custom scaling (by default square), good for passenger volumes
                edge_colour_fraction = (edge_fraction**(1/self.custom_edge_exponent))
            elif mode=='linear': #use linear scaling, good for distance to travel in smaller maps
                edge_colour_fraction = edge_fraction
            #convert node_colour_fraction to RGB, blue at 0, green at 0.3, red at 1

            midpoint = 0.3 #midpoint of colour scale is green
            if edge_colour_fraction<=midpoint:
                #colour scale is from blue to green
                green = edge_colour_fraction/midpoint
                blue =  1-green
                red = 0
            elif edge_colour_fraction>midpoint:
                #colour scale is from green to red
                red = (edge_colour_fraction-midpoint)/(1-midpoint)
                green = 1-red
                blue = 0

            #convert 24 bit RGB colour to the hex format expected by tkinter
            self.edge_colours[i] = RGB_TO_TK_HEX(int(red*255),int(green*255),int(blue*255))

    #FUNCTIONS CONTROLLING RENDERING OF NODES/EDGES
    #wrapper that recalculates node sizes and colours, and redraws the nodes
    def update_nodes(self):
        self.set_node_sizes()
        self.set_node_colours()
        self.render_graph()

     #wrapper that recalculates edge sizes and colours, and redraws the edges
    def update_edges(self):
```

```
        self.set_edge_widths()
        self.set_edge_colours()
        self.render_graph()

    #update text rendering next to nodes without changing the node whose information we are using (eg distance to/from that node)
    def update_text_same_node(self):
        if self.nodes_numeric_overlay_mode == 'node_total':
        #if we are overlaying based on total traffic too/from node, the key node does not matter, so we can display info without it
            self.text_total_passengers_node() #replace with new info about total traffic too/from a node
        elif self.nodes_numeric_overlay_mode == 'waiting_passengers':
            self.text_waiting_passengers_node() #replace with new info about passengers waiting at a node

        #otherwise don't update if no node-specific text was being displayed in the first place
        elif self.last_node_left_click_index == -1:
            self.erase_all_nodes_text('below') #erase all text already displayed
        else:
            last_click_index = self.last_node_left_click_index
            self.erase_all_nodes_text('below') #erase all text already displayed
            self.last_node_left_click_index = -1 #set this to -1 so update_nodes_viewing_mode correctly renders with a different mode (note keep this here for redunancy in case end up removing the reset from erase_all_nodes_text)
            self.update_nodes_viewing_mode_left_click(last_click_index) #update the render
            self.last_node_left_click_index = last_click_index #set last left click index back to it's previous value so we can still remove info by clicking on that node again

    #update the display relating to nodes in response to a left click
    def update_nodes_viewing_mode_left_click(self, left_click_index):
        if self.nodes_numeric_overlay_mode == 'node_total':
```

```
                self.text_total_passengers_node() #display the total number of passengers going too/from all nodes
        elif self.last_node_left_click_index == left_click_index: #if the same node has been clicked on again
            self.erase_all_nodes_text('below') #reset all text
            self.last_node_left_click_index = -1
        else: #otherwise, display info text for new node
            if self.nodes_numeric_overlay_mode == 'no_info':
                self.erase_all_nodes_text('below') #reset all text, as not used in this mode
            elif self.nodes_numeric_overlay_mode == 'node_relative':
                self.text_passengers_node(left_click_index) #display the number of passengers going too/from this particular node
            elif self.nodes_numeric_overlay_mode == 'node_distance':
                self.text_journeys_node(left_click_index) #display the time taken to travel from this node too/from all other nodes

            self.last_node_left_click_index = left_click_index #record this was the last node we clicked on

    #UTILITY FUNCTIONS

    #prints a message to the console only if the logging level is at a certain level (default=1)
    def log_print(self,message,log_level=1):
        if self.verbose>=log_level:
            print(message)

    #update the control message board
    def message_update(self, string):
        self.message.config(text=string)

    def convert_lat_long_to_x_y(self, latitude, longitude):
        latitude_offset = latitude-self.central_latitude
        longitude_offset = longitude-self.central_longitude
```

```
        y = self.canvas_center_y-(latitude_offset*self.pixels_per_degree) #we need to
            flip the offset along the y axis, as higher values mean further down (south)
             in canvas coordinates
        x = self.canvas_center_x+(longitude_offset*self.pixels_per_degree)
        return (x,y)

    #FUNCTIONS TO IMPORT DATA NEEDED FOR THE NETWORK

    #extract the list of nodes from a csv file into a python list, and calculate global
        geographical information for plotting
    def extract_nodes_graph(self):
        self.node_names = self.nodes_csv["Name"].to_list()
        node_positions = self.nodes_csv["Location"].to_list()
        self.node_latitudes = []
        self.node_longitudes = []
        for position in node_positions:
            latitude,longitude = n.extract_coordinates(position)
            self.node_latitudes.append(latitude)
            self.node_longitudes.append(longitude)

        #get the minimum/maximum longitude and latitude
        min_latitude = min(self.node_latitudes)
        max_latitude = max(self.node_latitudes)
        min_longitude = min(self.node_longitudes)
        max_longitude = max(self.node_longitudes)
        self.central_latitude = (min_latitude + max_latitude)/2
        self.central_longitude = (min_longitude + max_longitude)/2
        #now determine conversion factors between coordinates and pixels
        #note the canvas needs to be created first
        range_latitude = max_latitude-min_latitude
        range_longitude = max_longitude-min_longitude
        #extract the scaling factor between the canvas and the real world
        pixels_per_degree_vertical = (self.canvas_height-(self.max_node_radius*4))/
            range_latitude
        pixels_per_degree_horizontal = (self.canvas_width-(self.max_node_radius*4))/
            range_longitude
        #the lower value is the limiting factor for an undistorted map
```

```python
        self.pixels_per_degree = min(pixels_per_degree_vertical, pixels_per_degree_horizontal)

    #extract the list of edges from a csv file into a python list, and calculate global geographical information for plotting
    #this needs to be run after nodes have been extracted so start/end node index assignment can be done
    def extract_edges_graph(self):
        edge_starts = self.edges_csv["Start"].to_list()
        edge_ends = self.edges_csv["End"].to_list()
        self.edge_names = [] #name of the edge from start to end
        self.edge_reverse_names = [] #name of the edge from end to start
        num_edges = len(edge_starts)#for the purpose of plotting, a bidirectional edge is one edge
        #find the index of edge starts and ends in the list of nodes
        self.edge_start_indices = []
        self.edge_end_indices = []
        for i in range(num_edges):
            #get the start index
            try:
                start_index = self.node_names.index(edge_starts[i])
            except ValueError:
                warnings.warn('edge start ', edge_starts[i],' not present in list of node names')
                start_index = -1 #this will cause a crash later (by design), as our program a non-existent start node

            #get the end index
            try:
                end_index = self.node_names.index(edge_ends[i])
            except ValueError:
                warnings.warn('edge end ', edge_ends[i],' not present in list of node names')
                end_index = -1 #this will cause a crash later (by design), as our program contains a non-existent end node

            self.edge_names.append(edge_starts[i] + ' to ' + edge_ends[i])
```

```python
                self.edge_reverse_names.append(edge_ends[i] + ' to ' + edge_starts[i])
                self.edge_start_indices.append(start_index)
                self.edge_end_indices.append(end_index)

        self.edge_canvas_ids = ['blank']*num_edges #store edge canvas ids in a list so we can delete them later, 'blank' indicates they have not yet been created
        self.edge_widths = [self.default_edge_width]*num_edges #store the default width of every edge
        self.edge_colours = [self.default_edge_colour]*num_edges #store the default colour of every edge
        self.edge_arrows = [tk.NONE]*num_edges #by default there will be no arrows on an edge

    #calculate information about position of nodes
    def calculate_node_position(self):
        num_nodes = len(self.node_names)
        self.nodes_x = []
        self.nodes_y = []
        self.nodes_radii = [self.default_node_radius]*num_nodes #default size for nodes
        self.nodes_colour = [self.default_node_colour]*num_nodes #default
        self.node_below_text_ids = ['blank']*num_nodes  #canvas ids for text which could be displayed below all nodes
        self.node_above_text_ids = ['blank']*num_nodes  #canvas ids for text which could be displayed above all nodes
        self.node_canvas_ids = ['blank']*num_nodes #canvas ids for the nodes themsleves
        for i in range(num_nodes):
            x,y = self.convert_lat_long_to_x_y(self.node_latitudes[i],self.node_longitudes[i])
            self.nodes_x.append(x)
            self.nodes_y.append(y)
        #original copy of node position, so that scaling can be calculated relative to the original values
        self.nodes_x_original = self.nodes_x
        self.nodes_y_original = self.nodes_y

    #calculate the midpoint of edges, used for plotting overlay text on edges
    #this needs to be done after node positions are calculated
```

```
    def calculate_edges_midpoints(self):
        num_edges = len(self.edge_names)
        self.edges_midpoint_x = []
        self.edges_midpoint_y = []
        self.edge_text_ids = ['blank']*num_edges  #canvas ids for text which could be displayed next to all edges
        for i in range(num_edges):
            #extract the location of the nodes which the edge connects
            edge_start_index = self.edge_start_indices[i]
            edge_end_index = self.edge_end_indices[i]
            #and then calculate the midpoint of the edge
            edge_midpoint_x = (self.nodes_x[edge_start_index] + self.nodes_x[edge_end_index])/2
            edge_midpoint_y = (self.nodes_y[edge_start_index] + self.nodes_y[edge_end_index])/2
            #and store this calculated value in a list
            self.edges_midpoint_x.append(edge_midpoint_x)
            self.edges_midpoint_y.append(edge_midpoint_y)

        #original copy of edge midpoints, so that scaling can be calculated relative to the original values
        self.edges_midpoint_x_original = self.edges_midpoint_x
        self.edges_midpoint_y_original = self.edges_midpoint_y

    #calculate the position, colour and size of vehicles
    def calculate_vehicle_position(self):
        #as vehicle quantities change from timestep to timestep, need to delete old vehicles first
        self.derender_vehicles()
        num_vehicles = len(self.sim_vehicles_current_names)
        self.sim_vehicles_current_x = []
        self.sim_vehicles_current_y = []
        self.sim_vehicles_current_length = [self.default_vehicle_length]*num_vehicles
        self.set_vehicle_colours() #set the vehicle colour based on the choosen mode
        self.vehicle_canvas_ids = ['blank']*num_vehicles #canvas ids for the nodes themsleves
        for i in range(num_vehicles):
```

```
            x,y = self.convert_lat_long_to_x_y(self.sim_vehicles_current_latitudes[i], self.sim_vehicles_current_longitudes[i])
            x,y = self.apply_accumlated_zoom(x,y)#apply accumulated zoom to new vehicle objects
            self.sim_vehicles_current_x.append(x)
            self.sim_vehicles_current_y.append(y)
        self.sim_vehicles_current_x_original = self.sim_vehicles_current_x
        self.sim_vehicles_current_y_original = self.sim_vehicles_current_y

    #function to set the colour of vehicles
    def set_vehicle_colours(self):
        num_vehicles = len(self.sim_vehicles_current_names)
        self.sim_vehicles_current_colour = [self.default_vehicle_colour]*num_vehicles
        if self.vehicle_colour_type == "constant":
            pass #default colours have already been set
        elif self.vehicle_colour_type == "crowding":
            self.calculate_vehicle_colours_crowding(self.sim_vehicles_current_passengers, self.vehicle_seated_capacity, self.vehicle_standing_capacity)
        else:
            #default to the default colour, which has already been set
            message = "INVALID COLOUR TYPE " + self.vehicle_colour_type + "\n COLOUR SET TO DEFAULT"
            self.log_print(message)

    #FUNCTIONS PERFORMING ACTUAL RENDERING
    #derender displayed vehicles
    def derender_vehicles(self,override=False):
        #delete all existing vehicles
        #overide option allows the function to operate even simulation_view_flag is false
        if self.simulation_view_flag==True or override==True:
            num_vehicles_old = len(self.vehicle_canvas_ids)
            for i in range(num_vehicles_old):
                if self.vehicle_canvas_ids[i]!='blank':
                        #delete the old oval object if one exists
                        self.canvas.delete(self.vehicle_canvas_ids[i])

```

```python
    #derender the text produced by hovering over a vehicle
    def derender_hover_vehicle_text(self):
        if self.index_vehicle_text_popup ==-1:
            pass #there are no vehicle hover text and hence no need to derender it
        else:
            self.canvas.delete(self.text_id_vehicle_lower)
            self.canvas.delete(self.text_id_vehicle_upper)

    #rerender the text after the vehicle has been moved/zoomed in/out
    def render_hover_vehicle_text(self):
        if self.index_vehicle_text_popup ==-1:
            pass #there are no vehicle hover text and hence no need to render it
        else:
            x = self.sim_vehicles_current_x[self.index_vehicle_text_popup]
            y = self.sim_vehicles_current_y[self.index_vehicle_text_popup]
            vehicle_name = self.sim_vehicles_current_names[self.index_vehicle_text_popup
                ]
            lower_text = self.sim_vehicles_current_passengers[self.
                index_vehicle_text_popup]
            self.text_id_vehicle_upper = self.canvas.create_text(x,y-30,text=
                vehicle_name, state=tk.DISABLED)
            self.text_id_vehicle_lower = self.canvas.create_text(x,y-15,text=lower_text,
                state=tk.DISABLED)

    #derender edge text created by hovering
    def derender_hover_edge_text(self):
        if self.text_id_line_end != -1: #if such text exists
            self.canvas.delete(self.text_id_line_end)
            self.canvas.delete(self.text_id_line_start)
            self.text_id_line_start =  -1
            self.text_id_line_end = -1

    #needs to be run after edges have been extracted and nodes have been drawn to work
        correctly
    def render_edges(self):
        self.derender_hover_edge_text()#derender additional edge text if it exists
        num_edges = len(self.edge_start_indices)
```

```
        for i in range(num_edges):
            start_index = self.edge_start_indices[i]
            end_index = self.edge_end_indices[i]
            start_x = self.nodes_x[start_index]
            start_y = self.nodes_y[start_index]
            end_x = self.nodes_x[end_index]
            end_y = self.nodes_y[end_index]
            colour = self.edge_colours[i]
            width = int(self.edge_widths[i]) #interesting thing about tkinter, circles can have non-integer sizes but lines need integer sizes
            edge_arrow = self.edge_arrows[i]
            #end_size = self.nodes_radii[end_index] #unused, we draw nodes over edges so no need to crop the edges
            #print('width ', width)
            #print('activewidth ',width+self.active_width_addition)
            if self.edge_canvas_ids[i]!='blank':
                #delete the old line object if one exists
                self.canvas.delete(self.edge_canvas_ids[i])

            id = self.canvas.create_line(start_x, start_y, end_x, end_y, fill=colour, width=width, activewidth=width+self.active_width_addition, arrow=edge_arrow) #draw a line to represent the edge
            self.canvas.tag_bind(id, '<Enter>', self.edge_enter) #some information about the start and end nodes will be displayed when we mouse over an edge
            self.canvas.tag_bind(id, '<Leave>', self.edge_leave) #this information will stop being displayed when the mouse is no longer over the node
            self.edge_canvas_ids[i] = id

    #draw the nodes on the canvas
    def render_nodes(self):
        num_nodes = len(self.node_names)
        for i in range(num_nodes):
            #extract data
            x = self.nodes_x[i]
            y = self.nodes_y[i]
            radius = self.nodes_radii[i]
            colour = self.nodes_colour[i]
```

```
            try:
                self.canvas.delete(self.text_id) #delete the text popup if one exists
            except AttributeError:
                pass #if it does not exist, don't delete it
            #delete all old node objects
            if self.node_canvas_ids[i]!='blank':
                #delete the old oval object if one exists
                self.canvas.delete(self.node_canvas_ids[i])
            id = self.canvas.create_oval(x-radius,y-radius,x+radius,y+radius,fill=colour
                ) #draw a circle to represent the node
            self.canvas.tag_bind(id,'<Enter>',self.node_enter) #some information about
                the node will be displayed when the mouse is hovered over it
            self.canvas.tag_bind(id,'<Leave>',self.node_leave) #this information will
                stop being displayed when the mouse is no longer over the node
            self.canvas.tag_bind(id,'<Button-1>',self.node_left_click) #depending on
                gui_mode, information about the nodes relationship to other nodes will
                be displayed
            self.canvas.tag_bind(id,'<Button-2>',self.node_right_click) #depending on
                gui_mode, information about the nodes relationship to other nodes will
                be displayed
            self.node_canvas_ids[i] = id #store the id so we can delete the object later

    #draw the vehicle objects on the canvas
    def render_vehicles(self):
        num_vehicles = len(self.sim_vehicles_current_names)
        self.derender_hover_vehicle_text() #remove existing vehicle hover text
        #loop through all the current vehicles
        for i in range(num_vehicles):
            #extract data
            x = self.sim_vehicles_current_x[i]
            y = self.sim_vehicles_current_y[i]
            length = self.sim_vehicles_current_length[i]
            colour = self.sim_vehicles_current_colour[i]
            #delete all old vehicle objects
            if self.vehicle_canvas_ids[i]!='blank':
                #delete the old rectangle object if one exists
                self.canvas.delete(self.vehicle_canvas_ids[i])
```

```
            id = self.canvas.create_rectangle(x-length,y-length,x+length,y+length, fill=colour)
            self.canvas.tag_bind(id,'<Enter>',self.vehicle_enter) #some information about the vehicle will be displayed when the mouse is hovered over it
            self.canvas.tag_bind(id,'<Leave>',self.vehicle_leave) #this information will be displayed when the mouse is no longer over the vehicle
            self.canvas.tag_bind(id,'<Button-1>',self.vehicle_left_click) #this information will be displayed when the mouse is no longer over the vehicle
            self.canvas.tag_bind(id,'<Button-2>',self.vehicle_right_click) #this information will be displayed when the mouse is no longer over the vehicle
            #add code to display info about the vehicle when we hover over it
            self.vehicle_canvas_ids[i] = id #store the id so we can delete the object later

        self.render_hover_vehicle_text() #recreate old vehicle hover text at the new location

    #combination of render nodes and render edges, in correct order to prevent edges spawning over nodes
    def render_graph(self):
        self.render_edges()
        self.render_nodes()
        if self.simulation_run_flag == True:
            self.render_vehicles() #render vehicles if we are in simulation view mode

    #stop displaying all the nodes and edges
    def erase_network_graph(self):
        self.derender_hover_edge_text()
        #erase all edges
        for id in self.edge_canvas_ids:
            if id!='blank':
                self.canvas.delete(id)
        #erase all nodes
        for id in self.node_canvas_ids:
            if id!='blank':
```

```
                self.canvas.delete(id)

    #EVENT HANDLERS (eg clicking, hovering) FOR CANVAS NODES

    #event for when we mouse over a node, create a text box revealling node name and (
        planned) number of waiting passengers
    def node_enter(self,event):
        event_id = event.widget.find_withtag('current')[0]
        id_index = self.node_canvas_ids.index(event_id)
        node_name = self.node_names[id_index]
        self.log_print('node viewed ' + node_name)
        x = self.nodes_x[id_index]
        y = self.nodes_y[id_index]
        display_text = node_name
        self.text_id = self.canvas.create_text(x,y-15,text=display_text,state=tk.
            DISABLED) #create a text popup, which is not interactive

    #event for when the mouse leaves a node, remove the text box
    def node_leave(self,event):
        event_id = event.widget.find_withtag('current')[0]
        id_index = self.node_canvas_ids.index(event_id)
        node_name = self.node_names[id_index]
        self.log_print('node left ' + node_name)
        self.canvas.delete(self.text_id) #delete the text popup from node_enter

    #event for when we left-click on a node, outcome will depend on viewing mode
    def node_left_click(self,event):
        event_id = event.widget.find_withtag('current')[0]
        id_index = self.node_canvas_ids.index(event_id) #get the index of the node which
             has been clicked on
        if self.last_node_right_click_index !=-1: #if a node has been right clicked on
            self.reset_edges_plot() #remove any old route
            self.plot_path_nodes(id_index,self.last_node_right_click_index,text_nodes=
                False,arrows=True) #draw a path from the left clicked node to the right
                clicked node

```

```
            self.update_nodes_viewing_mode_left_click(id_index) #update the viewing mode due
                 to the click
            #rerender the nodes to be of the correct size after the new click
            self.update_nodes()

        #event for when we right-click on a node
        def node_right_click(self,event):
            event_id = event.widget.find_withtag('current')[0]
            id_index = self.node_canvas_ids.index(event_id) #get the index of the node which
                 has been clicked on
            if id_index == self.last_node_left_click_index: #right clicking on a node we
                just left clicked on will do nothing for now
                pass
            elif self.last_node_left_click_index == -1: #as will right clicking if no left
                click has occured
                pass
            else:
                self.reset_edges_plot() #remove any old route
                self.plot_path_nodes(self.last_node_left_click_index,id_index,text_nodes=
                    False,arrows=True) #draw a path from the left clicked node to the right
                    clicked node
                self.render_graph() #re-render the network
                self.last_node_right_click_index = id_index

        #EVENT HANDLERS FOR CANVAS EDGES

        #event for when we mouse over an edge, display text boxes above connected nodes
        def edge_enter(self,event):
            event_id = event.widget.find_withtag('current')[0]
            id_index = self.edge_canvas_ids.index(event_id)
            #find the nodes at the ends of the edge
            start_index = self.edge_start_indices[id_index]
            end_index = self.edge_end_indices[id_index]
            #find the x and y positions of these nodes
            start_x = self.nodes_x[start_index]
            start_y = self.nodes_y[start_index]
            end_x = self.nodes_x[end_index]
```

```python
        end_y = self.nodes_y[end_index]
        #decide on the text popup above each node
        display_text_start = self.node_names[start_index]
        display_text_end = self.node_names[end_index]
        #create the text popups, which are not interactive
        if self.text_id_line_start != -1:
            #delete any existing popups
            self.canvas.delete(self.text_id_line_start)
            self.canvas.delete(self.text_id_line_end)
        self.text_id_line_start = self.canvas.create_text(start_x, start_y-15, text=display_text_start, state=tk.DISABLED)
        self.text_id_line_end = self.canvas.create_text(end_x, end_y-15, text=display_text_end, state=tk.DISABLED)


    #event for when we mouse away from an edge
    def edge_leave(self, event):
        self.derender_hover_edge_text() #delete any hovering text related to the edge

    #event for when we mouse over a vehicle
    def vehicle_enter(self, event):
        event_id = event.widget.find_withtag('current')[0]
        id_index = self.vehicle_canvas_ids.index(event_id)
        vehicle_name = self.sim_vehicles_current_names[id_index]
        #delete hover text if it exists
        self.derender_hover_vehicle_text()
        #create new text popups
        self.index_vehicle_text_popup = id_index #record the index of the vehicle whose text popup we are creating
        self.name_vehicle_text_popup = vehicle_name #and also record the name, this is used to handle situations where the lists of vehicles changes
        self.render_hover_vehicle_text()


    #event for when we mouse away from a vehicle
    def vehicle_leave(self, event):
        event_id = event.widget.find_withtag('current')[0]
```

```
            id_index = self.vehicle_canvas_ids.index(event_id)
            #at the moment, we don't actually do anything here as we still want to display
                info about the vehicle when we are hovering over it

        #event for when we left click a vehicle
        def vehicle_left_click(self,event):
            event_id = event.widget.find_withtag('current')[0]
            id_index = self.vehicle_canvas_ids.index(event_id)
            #placeholder for future functionality

        #event for when we right click a vehicle
        def vehicle_right_click(self,event):
            event_id = event.widget.find_withtag('current')[0]
            id_index = self.vehicle_canvas_ids.index(event_id)
            #right clicks will reset the vehicle popup text rendering
            self.derender_hover_vehicle_text()
            self.index_vehicle_text_popup = -1
            self.name_vehicle_text_popup = -1
            #placeholder for future functionality

        #GENERAL CANVAS EVENT HANDLERS (FOR SCROLLING and ZOOMING IN/OUT)
        #zoom in/out
        def zoom_canvas(self,event):
            #get position of mouse during scroll
            mouse_x = self.canvas.canvasx(event.x)
            mouse_y = self.canvas.canvasy(event.y)
            #print('mouse x ',mouse_x,' mouse y ',mouse_y)
            zoom_delta = 0.01*event.delta #zoom is in proportion to scroll wheel direction
                and magnitude
            self.current_zoom = self.current_zoom*(1+zoom_delta) #update the accumulated
                zoom level
            self.current_zoom_offset_x = self.current_zoom_offset_x*(1+zoom_delta) - mouse_x
                *zoom_delta#calculate the new offset for x
            self.current_zoom_offset_y = self.current_zoom_offset_y*(1+zoom_delta) - mouse_y
                *zoom_delta#calculate the new offset for y
            self.apply_correct_zoom(zoom_delta,mouse_x,mouse_y) #perform the zoom on all
                objects in an image
```

```

    #recreate existing objects in the correctly zoomed position
    def apply_correct_zoom(self, zoom_delta, mouse_x, mouse_y):
        #update the graph
        self.recalculate_nodes_position(zoom_delta, mouse_x, mouse_y)
        self.recalculate_edge_midpoints(zoom_delta, mouse_x, mouse_y)
        if self.simulation_run_flag == True: #only recalculate vehicle position if vehicles exists
            self.recalculate_vehicle_position(zoom_delta, mouse_x, mouse_y)
        self.render_graph()
        self.node_names_update() #update the rendering of node names
        #update text overlays if simulation has been setup
        if self.simulation_setup_flag:
            self.update_text_same_node()
            self.generate_edge_overlay_text()

    #apply the accumulated zoom to newly created objects
    def apply_accumlated_zoom(self, x, y):
        new_x = (x*self.current_zoom)+(self.current_zoom_offset_x)
        new_y = (y*self.current_zoom)+(self.current_zoom_offset_y)
        return new_x, new_y

    #recalculate all node positions in response to the zoom action
    def recalculate_nodes_position(self, zoom_delta, mouse_x, mouse_y):
        num_nodes = len(self.nodes_x)
        #recalculate the position of all nodes
        for i in range(num_nodes):
            new_x, new_y = self.recalculate_zoom_position(self.nodes_x[i], self.nodes_y[i], zoom_delta, mouse_x, mouse_y)
            self.nodes_x[i] = new_x
            self.nodes_y[i] = new_y

    #recalculate the midpoint of all edges in response to zooming
    def recalculate_edge_midpoints(self, zoom_delta, mouse_x, mouse_y):
        num_edges = len(self.edges_midpoint_x)
        #recalculate the position of all edge midpoints
        for i in range(num_edges):
```

```
            new_x,new_y = self.recalculate_zoom_position(self.edges_midpoint_x[i],self.
                edges_midpoint_y[i],zoom_delta,mouse_x,mouse_y)
            self.edges_midpoint_x[i] = new_x
            self.edges_midpoint_y[i] = new_y

    #recalculate the position of all vehicles in response to zooming
    def recalculate_vehicle_position(self,zoom_delta,mouse_x,mouse_y):
        num_vehicles = len(self.sim_vehicles_current_names)
        #recalculate the position of all vehicles
        for i in range(num_vehicles):
            new_x,new_y = self.recalculate_zoom_position(self.sim_vehicles_current_x[i],
                self.sim_vehicles_current_y[i],zoom_delta,mouse_x,mouse_y)
            self.sim_vehicles_current_x[i] = new_x
            self.sim_vehicles_current_y[i] = new_y

    #recalculate the position of an object to be correct under the new zoom regime
    def recalculate_zoom_position(self,x,y,zoom_delta,mouse_x,mouse_y):
        new_x = x*(1+zoom_delta)-(mouse_x*zoom_delta)
        new_y = y*(1+zoom_delta)-(mouse_y*zoom_delta)
        return new_x,new_y

    #define pan function
    def pan_start(self,event):
        #get position of mouse at start of pan
        #mouse_x = int(self.canvas.canvasx(event.x))
        #mouse_y = int(self.canvas.canvasy(event.y))
        #print('mouse x ',mouse_x,' mouse y ',mouse_y)
        self.canvas.scan_mark(event.x, event.y) #record the position of start of scan

    #define scan function
    def pan_end(self,event):
        #get position of mouse at start of pan
        #mouse_x = int(self.canvas.canvasx(event.x))
        #mouse_y = int(self.canvas.canvasy(event.y))
        #print('mouse x ',mouse_x,' mouse y ',mouse_y)
        self.canvas.scan_dragto(event.x, event.y,gain=self.scroll_gain) #record the
            position of start of scan
```

```python

    #FUNCTIONS TO GENERATE INFO TEXT ABOVE NODES

    #display the number of passengers travelling to/from a clicked node to all other
        nodes (per hour as currently setup) as text above the nodes
    def text_passengers_node(self,key_node_index):
        if self.from_node:
            trips = self.sim_network.origin_destination_trips[key_node_index,:] #extract
                number of trips starting from this node
        else:
            trips = self.sim_network.origin_destination_trips[:,key_node_index] #extract
                number of trips going to this node

        self.display_text_info_node(trips,where_mode='below',type_mode='float') #display
            the number of trips starting/ending at every other node

    #display the number of passengers travelling to/from a node to all other nodes
        combined (per hour as currently setup) as text above the nodes
    def text_total_passengers_node(self):
        if self.from_node:
            trips = np.sum(self.sim_network.origin_destination_trips,0)#extract number
                of trips starting from all nodes
        else:
            trips = np.sum(self.sim_network.origin_destination_trips,1) #extract number
                of trips ending at all nodes

        self.display_text_info_node(trips,where_mode='below',type_mode='float') #display
            the number of trips starting/ending at each node node

    #display the number of passengers waiting at a node
    def text_waiting_passengers_node(self):
        self.display_text_info_node(self.sim_node_current_passengers,where_mode='below',
            type_mode='int') #display the number of waiting passengers at each node

    #display the journey time from the clicked node to other nodes as text above the
        node
    def text_journeys_node(self,key_node_index):
```

```
        if self.from_node:
            times = self.sim_network.distance_to_all[key_node_index,:] #extract journey times starting at this node
        else:
            times = self.sim_network.distance_to_all[:,key_node_index] #extract journey times going to this node

        self.display_text_info_node(times,type_mode='integer',where_mode='below') #display journey times to/from every other node

    #perform the actual text rendering of text near all nodes
    #whether this happens above or below all nodes can be selected
    def display_text_info_node(self,info,where_mode='below',type_mode='text'):
        num_nodes = len(self.node_names)
        self.erase_all_nodes_text(mode=where_mode) #clear any old text
        if where_mode=='below':
            self.node_below_text_ids = ['blank']*num_nodes #create a container for the new text ids
        elif where_mode=='above':
            self.node_above_text_ids = ['blank']*num_nodes #create a container for the new text ids
        for i in range(num_nodes): #for every node
            node_x = self.nodes_x[i]
            node_y = self.nodes_y[i]
            this_info = info[i]
            if type_mode=='float':
                this_info = "{:.2f}".format(this_info) #floating point data
            elif type_mode=='integer':
                this_info = str(this_info) #integer data
            if where_mode=='below':
                self.node_below_text_ids[i] = self.canvas.create_text(node_x,node_y+15, text=this_info, state=tk.DISABLED, fill=self.default_node_text_colour) #create a text popup, which is not interactive
            elif where_mode=='above':
                self.node_above_text_ids[i] = self.canvas.create_text(node_x,node_y-15, text=this_info, state=tk.DISABLED, fill=self.default_node_text_colour) #create a text popup, which is not interactive
```

```

    #erase text displayed next to all nodes (eg num passengers/journey time)
    def erase_all_nodes_text(self,mode='both'):
        #self.last_node_left_click_index = -1 #we are deleting all nodes text, so reset
            if any nodes have been clicked
        #text to delete depends on mode
        if mode == 'above':
            text_ids = self.node_above_text_ids
        elif mode == 'below':
            text_ids = self.node_below_text_ids
        elif mode == 'both':
            text_ids = self.node_above_text_ids + self.node_below_text_ids
        #delete the selected text
        for id in text_ids:
            if id!='blank':
                self.canvas.delete(id)

    #FUNCTIONS TO GENERATE INFO TEXT ABOVE EDGES

    #get data about a specific edge from the network
    #valid types are "time" and "traffic"
    def get_edge_data(self,edge_name,type):
        index = self.sim_network.get_edge_index(edge_name)#get the index of the edge in
            the network data structure
        if type == 'time':
            data = self.sim_network.get_edge_time(edge_name)
        elif type == 'traffic':
            data = self.sim_network.get_edge_traffic(edge_name)
        return data

    def extract_data_edges(self,type):
        forward_edge_data = []
        reverse_edge_data = []
        for forward_edge_name in self.edge_names: #extract data from the forward edges
            forward_edge_data.append(self.get_edge_data(forward_edge_name,type))
        for reverse_edge_name in self.edge_reverse_names:
            reverse_edge_data.append(self.get_edge_data(reverse_edge_name,type))
```

```
        return forward_edge_data, reverse_edge_data

    #determine the actual text which will be displayed on the edges
    #if combine is true, the data will be added together for display
    def determine_edges_text(self,type,combine):
        (forward_edge_data,reverse_edge_data) = self.extract_data_edges(type) #extract
            forward and edge data
        edges_text = []
        num_edges = len(forward_edge_data)
        if combine:
            for i in range(num_edges):
                combined_data =  reverse_edge_data[i] + forward_edge_data[i]
                edges_text.append(format(combined_data,'.2f'))

        else:
            if self.edge_direction_mode == 'forward':
                for i in range(num_edges):
                    edges_text.append(format(forward_edge_data[i],'.2f'))
            elif self.edge_direction_mode == 'reverse':
                for i in range(num_edges):
                    edges_text.append(format(reverse_edge_data[i],'.2f'))
            elif self.edge_direction_mode == 'both':
                for i in range(num_edges):
                    edges_text.append(format(reverse_edge_data[i],'.2f') + '/' + format(
                        reverse_edge_data[i],'.2f'))

        return edges_text

    #generate and plot the overlay text for edges
    def generate_edge_overlay_text(self):
        if self.edges_numeric_overlay_mode == 'no_info':
            self.erase_all_edges_text() #delete any existing edge text
            return #exit the function, we don't need to do anything more
        elif self.edges_numeric_overlay_mode == 'distance':
            edges_text = self.determine_edges_text('time',False)
        elif self.edges_numeric_overlay_mode == 'traffic':
            edges_text = self.determine_edges_text('traffic',False)
```

```
            elif self.edges_numeric_overlay_mode == 'total_traffic':
                edges_text = self.determine_edges_text('traffic',True)
            #display the text previously generated
            self.display_text_info_above_edges(edges_text)

    def display_text_info_above_edges(self,info):
        num_edges = len(self.edge_names)
        self.erase_all_edges_text() #clear any old text
        self.edge_text_ids = ['blank']*num_edges #create a container for the new text
            ids
        for i in range(num_edges): #for every edge
            edge_x = self.edges_midpoint_x[i]
            edge_y = self.edges_midpoint_y[i]
            self.edge_text_ids[i] = self.canvas.create_text(edge_x,edge_y,text=info[i],
                state=tk.DISABLED, fill=self.default_edge_text_colour) #create a text
                popup, which is not interactive

    #render edge names
    def render_edge_names(self):
        self.display_text_info_above_edges(self.edge_names)

    #erase text displayed next to all edges
    def erase_all_edges_text(self):
        self.last_edge_left_click_index = -1 #we are deleting all nodes text, so reset
            if any edges have been clicked
        for id in self.edge_text_ids:
            if id!='blank':
                self.canvas.delete(id)

    #FUNCTIONS TO GENERATE INFO TEXT ABOVE VEHICLES


    #FUNCTIONS TO PLOT A PATH BETWEEN TWO NODES

    #extract the path between two node based on their indices
    def extract_path_node_indices(self,start_node_index,end_node_index):
        edges_path = self.sim_network.paths_to_all[start_node_index][end_node_index]
```

```
        return edges_path

    #extract the path between two nodes
    def extract_path_nodes(self,start_node,end_node):
        start_id = self.node_names.index(start_node) #get the id's of the starting node
        end_id = self.node_names.index(end_node) #and the ending node
        edges_path = self.extract_path_node_indices(start_id,end_id)
        return edges_path

    #reset edge names and colours to their default values
    def reset_edges_plot(self):
        num_edges = len(self.edge_names)
        for i in range(num_edges):
            self.edge_colours[i] = self.default_edge_colour
            self.edge_widths[i] = self.default_edge_width
            self.edge_arrows[i] = tk.NONE

    #plot the path between two nodes
    def plot_path_nodes(self,start_node,end_node,text_nodes=True,arrows='auto'):
        #if text_nodes = True, we select the path using the verbose names of the nodes, rather than just their index
        if text_nodes:
            edges_path = self.extract_path_nodes(start_node,end_node)
        else:
            edges_path = self.extract_path_node_indices(start_node,end_node)
        #will arrows be present on plotted path
        if arrows=='auto':
            arrows = self.path_edge_arrows #by default, choose initally defined default option

        for edge_name in edges_path:
            #go through all the edges in the edges path
            try:
                #if the edge is from start to finish
                edge_index = self.edge_names.index(edge_name)
                reverse = False
            except ValueError:
```

```
                #if the edge is from finish to start
                try:
                    edge_index = self.edge_reverse_names.index(edge_name)
                    reverse = True
                except ValueError:
                    #edge is in neither list
                    warnings.warn('edge ',edge_name,' not present in list of edges')
                    continue

            #now update edge names and colours for nodes on the path
            self.edge_colours[edge_index] = self.path_edge_colour
            self.edge_widths[edge_index] = self.path_edge_width
            if arrows==True:#if we are plotting arrows
                if reverse: #draw an arrow pointing towards the starting node
                    self.edge_arrows[edge_index] = tk.FIRST
                else: #draw an arrow pointing away from the starting node
                    self.edge_arrows[edge_index] = tk.LAST
            else: #if we are not plotting arrows
                self.edge_arrows[edge_index] = tk.NONE #don't plot arrows


#EXTERNAL UTILITY FUNCTIONS

#convert 24bit RGB colour to the hex format used by tkinter
def RGB_TO_TK_HEX(red, green, blue):
    #convert to hex and remove leading 0x
    red_string = int_to_2hex(red) #extract from 3rd element in string to last element
    green_string = int_to_2hex(green)
    blue_string = int_to_2hex(blue)
    output_string = "#" + red_string + green_string + blue_string #combine components into the correct format
    return output_string

#converts integers to hexs of at least length 2(so can represent numbers 0-255)
def int_to_2hex(num):#converts an integer to a length 2 hex
    string = hex(num)[2:]
    #prepend 0's if hex is too short
```

```
    if len(string)==1:
        string = '0' + string
    elif len(string)==0:
        string = '00'

    return string

#utility which takes as input an edge name and produces the name an edge going between
    the same nodes but in the opposite direction
#note this utility cannot handle destinations where " to " is part of the name
def reverse_edge_name(edge_name):
    divider_string = ' to '
    divider_start = edge_name.find(divider_string) #start of the division between origin
        and destination
    origin = edge_name[0:divider_start] #extract origin name
    destination = edge_name[divider_start+len(divider_string):] #and destination name
    output_string = destination + divider_string + origin #create the reversed string
    return output_string
```

A2 Appendix B: CSV Listings

The example CSV files which store information related to the Sydney Trains system are included below. They are also available online at https://github.com/henryc47/Thesis_Public_Transport_Optimisation

Name	Daily Passengers	Location		
Berowra	1,635	-33.62344878916775, 151.15302817140895		
Mt Ku-Ring Gai	285	-33.653168045041994, 151.1369620382525		
Mount Colah	610	-33.67151166126941, 151.11506206462323		
Asquith	1,765	-33.688765392444296, 151.1082948589636		
Hornsby	13,320	-33.703561248359705, 151.09834807114868		
Waitara	3,875	-33.70999945059616, 151.1044472994591		
Wahroonga	2,395	-33.71739484320278, 151.11696214555374		
Warrawee	1,525	-33.72428864708873, 151.12176539846928		
Turramurra	4,585	-33.73236019237686, 151.1283609772862		
Pymble	2,740	-33.744613802711505, 151.14199267215804		
Gordon	7,365	-33.7558028491355, 151.1543372950995		
Killara	2,520	-33.76548147595771, 151.1617059804787		
Lindfield	3,935	-33.77561096974262, 151.1692050773864		
Roseville	2,915	-33.78416016738154, 151.17732885431892		
Chatswood	28,475	-33.79800409081103, 151.18088752427778		
Atarmon	6,165	-33.80886234435643, 151.18514595332843		
St Leonards	19,000	-33.82241547827582, 151.1941691104307		
Wollstonecraft	2,820	-33.83197064417154, 151.1917992288703		
Waverton	2,320	-33.83787525167453, 151.1975696042687		
North Sydney	33,595	-33.84111674463287, 151.2074232053009		
Milsons Point	8,005	-33.84586423518028, 151.21189670590542		
Wynyard	86,575	-33.86567784248656, 151.20613531940973		
Town Hall	113,405	-33.87325332874709, 151.2070346520971		
Central	116,985	-33.88272081009569, 151.20651688008556		
Redfern	31,150	-33.8916415223102, 151.19889132824312		
Macdonaldtown	1,250	-33.89660953844508, 151.1863105843083		
Newtown	11,535	-33.89785723557127, 151.17960668383392		
Stanmore	3,750	-33.894521966243104, 151.1639013113274		
Petersham	3,675	-33.89380991985312, 151.1550944170816		
Lewisham	3,100	-33.8931554842807, 151.14742403918422		
Summer Hill	3,875	-33.8902869562006, 151.13878572329725		
Ashfield	13,415	-33.88750487240862, 151.1258969206334		
Croydon	2,675	-33.88310564104322, 151.11529122631816		
Burwood	20,090	-33.87712270561161, 151.10428538117588		
Strathfield	24,955	-33.871959812692666, 151.09447906025235		
Normanhurst	1,820	-33.72080742999822, 151.09708223431744		
Thornleigh	2,160	-33.731776847388396, 151.0783146625566		
Pennant Hills	3,530	-33.738060204121496, 151.07239623982645		
Beecroft	2,355	-33.749658601180364, 151.06638097218368		
Cheltenham	1,440	-33.755728446573414, 151.0785695490641		
Epping	12,750	-33.77275163450705, 151.08196604669925		
Eastwood	8,410	-33.79010957162102, 151.0820747017319		
Denistone	670	-33.799571292601186, 151.0867289605078		
West Ryde	4,730	-33.80734133275845, 151.09020593812863		
Meadowbank	5,150	-33.81597925104732, 151.0901065394409		

Name	Daily Passengers	Location		
Rhodes	10,920	-33.83056692462465, 151.087059914035		
Concord West	2,445	-33.84849381941757, 151.08561503951583		
North Strathfield	2,985	-33.858948158638206, 151.08819444387782		
Tallawong	1,250	-33.69155330414296, 150.90600934170322		
Rouse Hill	3,000	-33.6919607824674, 150.92437986563752		
Kellyville	4,000	-33.71353480888214, 150.93533635504676		
Bella Vista	5,500	-33.730548171620384, 150.94423493366276		
Norwest	3,000	-33.73445439047416, 150.96368928524674		
Hills Showgrounc	3,500	-33.72804671807569, 150.98686422990042		
Castle Hill	6,500	-33.731570892750014, 151.0074882877244		
Cherrybrook	4,000	-33.73669022333961, 151.03186193489964		
Macquarie Unive	12,645	-33.77719159428722, 151.11820153337777		
Macquarie Park	4,855	-33.785177552067616, 151.12835920719		
North Ryde	2,855	-33.794514553606504, 151.13802398846065		
Homebush	3,405	-33.86674687990985, 151.0860664946239		
Flemington	4,735	-33.86484084951063, 151.07022834253257		
Lidcombe	13,870	-33.86355273771468, 151.04477718428515		
Auburn	14,175	-33.8492527692842, 151.03279076577232		
Clyde	1,650	-33.83575044915596, 151.01695672825883		
Granville	6,525	-33.83284221081737, 151.0118652672295		
Harris Park	2,125	-33.82335567454209, 151.00776020985728		
Parramatta	46,475	-33.8172040474383, 151.00488617207003		
Westmead	7,700	-33.808491867627055, 150.98788461205788		
Wentworthville	4,320	-33.807176725579474, 150.97266104453269		
Pendle Hill	3,640	-33.80143962110018, 150.95636459037		
Toongabbie	2,860	-33.787374238353024, 150.95143000650742		
Seven Hills	7,320	-33.77437078486457, 150.93615031644134		
Blacktown	17,600	-33.76861961244888, 150.90741514512217		
Doonside	2,935	-33.76383974493935, 150.8692215857484		
Rooty Hill	3,335	-33.77147829931052, 150.8451610043804		
Mount Druitt	8,095	-33.769540437198934, 150.82009126293318		
St Mary's	5,045	-33.76206804300568, 150.7751573654032		
Werrington	1,200	-33.759190018348505, 150.75770677376744		
Kingswood	2,625	-33.75833822217043, 150.7205274705547		
Penrith	8,565	-33.75003869109963, 150.6958608438714		
Emu Plains	1,670	-33.74565372069082, 150.67185313408316		
Marayong	1,155	-33.746299808304, 150.9002814613991		
Quakers Hill	4,220	-33.72735112430899, 150.88629055353476		
Schofields	2,200	-33.70461149851124, 150.87393099473505		
Riverstone	960	-33.679129177404704, 150.86031064533603		
Vineyard	140	-33.65049690544507, 150.85113645867136		
Mulgrave	370	-33.626583677409045, 150.83048899597		
Windsor	955	-33.613814768730315, 150.8112628127668		
Claredon	95	-33.60857477874858, 150.78789305109353		
Richmond	1,210	-33.59890394880124, 150.75258486166138		

Name	Daily Passengers	Location		
Circular Quay	25,910	-33.8612093243693, 151.2106834162501		
St James	11,775	-33.87083044689528, 151.2104560743835		
Museum	14,095	-33.876422309916606, 151.20969253867463		
Bondi Junction	27,995	-33.89118914670372, 151.24842415995275		
Edgecliff	10,010	-33.879402150006236, 151.23638162028348		
Kings Cross	16,550	-33.87439504149759, 151.22254378721576		
Martin Place	25,125	-33.86812988958452, 151.21167026122666		
Erskineville	2,660	-33.89994513529435, 151.1856027380996		
St Peters	3,840	-33.90738045004097, 151.1803232907522		
Sydenham	6,450	-33.9146462177433, 151.1660313581001		
Marrickville	4,500	-33.91362315114232, 151.15318595974287		
Dulwich Hill	2,820	-33.9109433877177, 151.141319790257		
Hurlstone Park	1,545	-33.91031969889618, 151.1326078708523		
Canterbury	3,075	-33.91174986133984, 151.11870490719576		
Campsie	9,125	-33.9102308330109, 151.10360689668403		
Belmore	2,915	-33.91731422186011, 151.0887661279762		
Lakemba	4,340	-33.92022989671872, 151.07593461171248		
Wiley Park	1,850	-33.922802412187636, 151.06818904322694		
Punchbowl	2,915	-33.925532292764146, 151.05557054527156		
Bankstown	10,035	-33.91773969776826, 151.0346659284177		
Cabramatta	9,035	-33.89471305177015, 150.93928552649135		
Carramar	585	-33.884472583639976, 150.9614855631568		
Villawood	520	-33.881124070265024, 150.97602998288454		
Leightonfield	285	-33.8816750162578, 150.98524057051918		
Chester Hill	1,260	-33.883762853525795, 150.99986641742126		
Sefton	750	-33.885441168823675, 151.01136858033325		
Regents Park	1,400	-33.88308156370094, 151.02442562610412		
Berala	2,210	-33.871938397388895, 151.0326477276576		
Birrong	1,210	-33.89330972728165, 151.02419945021384		
Yagoona	1,710	-33.90689441015184, 151.02461301052014		
Tempe	1,655	-33.92377062227249, 151.15663588356122		
Wolli Creek	8,370	-33.92836912317192, 151.15349066164663		
Turrella	1,195	-33.92970064746019, 151.1401818059347		
Bardwell Park	1,290	-33.93142598100953, 151.1249853013555		
Bexley North	1,320	-33.93737393768401, 151.11353223932093		
Kingsgrove	2,605	-33.940440195129895, 151.10047072692342		
Beverly Hills	2,375	-33.9488256157126, 151.08116276322747		
Narwee	1,925	-33.94743426218551, 151.07037838973923		
Riverwood	4,290	-33.95124213013919, 151.05243540678197		
Padstow	3,670	-33.95175513950131, 151.0324428406625		
Revesby	5,030	-33.95230395858278, 151.01491866920873		
Panania	1,735	-33.95416289782418, 150.99820502254434		
East Hills	1,010	-33.96183544756428, 150.98476790560724		
Holsworthy	3,940	-33.96325846886507, 150.95667496141957		
Glenfield	6,500	-33.97233224125516, 150.89325416579675		

Name	Daily Passengers	Location		
Macquarie Fields	1,070	-33.984792230414406, 150.87943143175616		
Ingleburn	3,765	-33.997779474483536, 150.86454556123863		
Minto	3,625	-34.02733956843438, 150.8426276308612		
Leumeah	3,080	-34.050619090706, 150.8304436384795		
Campbelltown	6,390	-34.06408998196798, 150.81433840389832		
Macarthur	3,305	-34.071752988822716, 150.79718071327378		
Leppington	2,750	-33.95457522283649, 150.80801405726908		
Edmondson Park	2,500	-33.969129360404196, 150.85860447556655		
Casula	385	-33.94996476681788, 150.91224325439842		
Liverpool	9,910	-33.92441943788642, 150.92752686379757		
Warwick Farm	2,740	-33.91304965068138, 150.9353773924965		
Canley Vale	2,865	-33.88723057326514, 150.9437868837462		
Fairfield	7,815	-33.87248029315455, 150.9569269257351		
Yennora	1,215	-33.86482772766398, 150.9709466263343		
Guildford	3,230	-33.854192425478615, 150.98460933472938		
Merrylands	6,050	-33.83654978085011, 150.99277761852616		
Mascot	16,000	-33.9061503821023, 151.20260299809732		
Green Square	11,150	-33.923158591176986, 151.1874195365624		
Domestic	21,000	-33.93348309318093, 151.18109630453193		
International	15,000	-33.9351342417784, 151.16679665408324		
Arncliffe	2,805	-33.936257974954614, 151.1474664273147		
Banksia	1,380	-33.94527222419022, 151.14062456634431		
Rockdale	10,865	-33.95209099162781, 151.1367877563197		
Kogarah	12,420	-33.962564167453024, 151.13246516154817		
Carlton	3,170	-33.96820331053866, 151.12422499722862		
Allawah	3,150	-33.96951303098739, 151.11456263789506		
Hurstville	22,530	-33.96748835656106, 151.10244881908793		
Penshurst	3,630	-33.9660601446999, 151.0893307762238		
Mortdale	3,890	-33.97045767593273, 151.0812677960963		
Oatley	2,405	-33.98062016403029, 151.07909408437015		
Como	850	-34.00413677923535, 151.06793904856852		
Jannali	2,885	-34.015797346365865, 151.06459457369556		
Sutherland	7,795	-34.03148179795843, 151.05749184266477		
Cronulla	2,685	-34.05566769805897, 151.15147363091123		
Woolooware	1,395	-34.047654077402754, 151.14404544147098		
Caringbah	3,275	-34.0415030952828, 151.12260705323786		
Miranda	4,065	-34.036303583601295, 151.10267671108022		
Gymea	2,525	-34.034871574785534, 151.0855638174771		
Kirrawee	1,555	-34.03496058116302, 151.0717018177707		
Waterfall	480	-34.13449333126558, 150.99457837739095		
Heathcote	650	-34.088039455552526, 151.00825357047015		
Engadine	1,460	-34.067572780340726, 151.01486453766947		
Loftus	645	-34.04506781405283, 151.05133104346		
Olympic Park	3,160	-33.846300832273016, 151.0695022804459		

Start	End	Time	Bidirectional
Berowra	Mt Ku-Ring Gai	4	Yes
Berowra	Hornsby	10	Yes
Mt Ku-Ring Gai	Mount Colah	3	Yes
Mount Colah	Asquith	3	Yes
Asquith	Hornsby	3	Yes
Hornsby	Waitara	2	Yes
Waitara	Wahroonga	2	Yes
Wahroonga	Warrawee	2	Yes
Warrawee	Turramurra	2	Yes
Turramurra	Pymble	3	Yes
Pymble	Gordon	3	Yes
Gordon	Killara	2	Yes
Gordon	Chatswood	6	Yes
Killara	Lindfield	2	Yes
Lindfield	Roseville	2	Yes
Roseville	Chatswood	3	Yes
Chatswood	Atarmon	2	Yes
Chatswood	St Leonards	4	Yes
Atarmon	St Leonards	3	Yes
St Leonards	Wollstonecraft	3	Yes
St Leonards	North Sydney	6	Yes
Wollstonecraft	Waverton	2	Yes
Waverton	North Sydney	3	Yes
North Sydney	Milsons Point	2	Yes
Milsons Point	Wynyard	4	Yes
Wynyard	Town Hall	2	Yes
Town Hall	Central	3	Yes
Hornsby	Normanhurst	3	Yes
Hornsby	Epping	10	Yes
Normanhurst	Thornleigh	3	Yes
Thornleigh	Pennant Hills	2	Yes
Pennant Hills	Beecroft	3	Yes
Beecroft	Cheltenham	2	Yes
Cheltenham	Epping	3	Yes
Epping	Eastwood	2	Yes
Epping	Strathfield	9	Yes
Eastwood	Denistone	2	Yes
Eastwood	Rhodes	5	Yes
Denistone	West Ryde	2	Yes
West Ryde	Meadowbank	3	Yes
Meadowbank	Rhodes	2	Yes
Rhodes	Concord West	3	Yes
Rhodes	Strathfield	5	Yes
Concord West	North Strathfield	2	Yes
North Strathfield	Strathfield	2	Yes

Start	End	Time	Bidirectional
Central	Redfern	2	Yes
Central	Strathfield	12	Yes
Redfern	Macdonaldtown	3	Yes
Redfern	Strathfield	11	Yes
Redfern	Burwood	10	Yes
Redfern	Ashfield	8	Yes
Redfern	Newtown	4	Yes
Macdonaldtown	Newtown	2	Yes
Newtown	Stanmore	2	Yes
Newtown	Ashfield	6	Yes
Stanmore	Petersham	2	Yes
Petersham	Lewisham	2	Yes
Lewisham	Summer Hill	2	Yes
Summer Hill	Ashfield	2	Yes
Ashfield	Croydon	3	Yes
Ashfield	Burwood	4	Yes
Croydon	Burwood	2	Yes
Burwood	Strathfield	3	Yes
Tallawong	Rouse Hill	2	Yes
Rouse Hill	Kellyville	3	Yes
Kellyville	Bella Vista	2	Yes
Bella Vista	Norwest	3	Yes
Norwest	Hills Showground	2	Yes
Hills Showgrounc	Castle Hill	3	Yes
Castle Hill	Cherrybrook	2	Yes
Cherrybrook	Epping	6	Yes
Epping	Macquarie Univers	4	Yes
Macquarie Unive	Macquarie Park	2	Yes
Macquarie Park	North Ryde	2	Yes
North Ryde	Chatswood	5	Yes
Strathfield	Homebush	2	Yes
Homebush	Flemington	2	Yes
Flemington	Lidcombe	3	Yes
Lidcombe	Auburn	3	Yes
Auburn	Clyde	2	Yes
Clyde	Granville	2	Yes
Granville	Harris Park	3	Yes
Harris Park	Parramatta	2	Yes
Strathfield	Flemington	3	Yes
Strathfield	Lidcombe	5	Yes
Auburn	Granville	3	Yes
Granville	Parramatta	4	Yes
Strathfield	Parramatta	12	Yes
Parramatta	Westmead	3	Yes
Parramatta	Blacktown	9	Yes

Start	End	Time	Bidirectional
Westmead	Wentworthville	2	Yes
Westmead	Seven Hills	6	Yes
Wentworthville	Pendle Hill	3	Yes
Pendle Hill	Toongabbie	3	Yes
Toongabbie	Seven Hills	3	Yes
Seven Hills	Blacktown	3	Yes
Blacktown	Doonside	4	Yes
Doonside	Rooty Hill	3	Yes
Rooty Hill	Mount Druitt	3	Yes
Mount Druitt	St Mary's	3	Yes
St Mary's	Werrington	3	Yes
Werrington	Kingswood	3	Yes
Kingswood	Penrith	3	Yes
Penrith	Emu Plains	3	Yes
Blacktown	Mount Druitt	8	Yes
Mount Druitt	Penrith	8	Yes
Blacktown	Penrith	14	Yes
Blacktown	Marayong	3	Yes
Marayong	Quakers Hill	3	Yes
Quakers Hill	Schofields	3	Yes
Schofields	Riverstone	3	Yes
Riverstone	Vineyard	4	Yes
Vineyard	Mulgrave	3	Yes
Mulgrave	Windsor	3	Yes
Windsor	Claredon	3	Yes
Claredon	Richmond	4	Yes
Wynyard	Circular Quay	3	Yes
Circular Quay	St James	3	Yes
St James	Museum	2	Yes
Museum	Central	3	Yes
Bondi Junction	Edgecliff	3	Yes
Edgecliff	Kings Cross	2	Yes
Kings Cross	Martin Place	3	Yes
Martin Place	Town Hall	2	Yes
Redfern	Erskineville	3	Yes
Erskineville	St Peters	2	Yes
St Peters	Sydenham	3	Yes
Redfern	Sydenham	6	Yes
Redfern	Wolli Creek	8	Yes
Sydenham	Marrickville	3	Yes
Marrickville	Dulwich Hill	2	Yes
Dulwich Hill	Hurlstone Park	2	Yes
Hurlstone Park	Canterbury	2	Yes
Canterbury	Campsie	2	Yes
Campsie	Belmore	3	Yes

Start	End	Time	Bidirectional
Belmore	Lakemba	2	Yes
Lakemba	Wiley Park	2	Yes
Wiley Park	Punchbowl	2	Yes
Punchbowl	Bankstown	3	Yes
Marrickville	Campsie	6	Yes
Sydenham	Campsie	8	Yes
Campsie	Bankstown	8	Yes
Cabramatta	Carramar	3	Yes
Carramar	Villawood	2	Yes
Villawood	Leightonfield	1	Yes
Leightonfield	Chester Hill	2	Yes
Chester Hill	Sefton	2	Yes
Sefton	Regents Park	3	Yes
Regents Park	Berala	2	Yes
Berala	Lidcombe	3	Yes
Sefton	Birrong	4	Yes
Birrong	Yagoona	2	Yes
Yagoona	Bankstown	3	Yes
Birrong	Regents Park	3	Yes
Cabramatta	Sefton	7	Yes
Sefton	Lidcombe	6	Yes
Sydenham	Tempe	2	Yes
Tempe	Wolli Creek	2	Yes
Sydenham	Wolli Creek	3	Yes
Wolli Creek	Turrella	2	Yes
Wolli Creek	Kingsgrove	5	Yes
Wolli Creek	Revesby	11	Yes
Turrella	Bardwell Park	2	Yes
Bardwell Park	Bexley North	2	Yes
Bexley North	Kingsgrove	2	Yes
Kingsgrove	Beverly Hills	3	Yes
Kingsgrove	Revesby	8	Yes
Beverly Hills	Narwee	2	Yes
Narwee	Riverwood	3	Yes
Riverwood	Padstow	3	Yes
Padstow	Revesby	2	Yes
Revesby	Panania	2	Yes
Revesby	Glenfield	8	Yes
Panania	East Hills	2	Yes
East Hills	Holsworthy	3	Yes
Holsworthy	Glenfield	5	Yes
Glenfield	Macquarie Fields	2	Yes
Macquarie Fields	Ingleburn	3	Yes
Ingleburn	Minto	4	Yes
Minto	Leumeah	3	Yes

Start	End	Time	Bidirectional
Leumeah	Campbelltown	3	Yes
Campbelltown	Macarthur	3	Yes
Leppington	Edmondson Park	5	Yes
Edmondson Park	Glenfield	4	Yes
Glenfield	Casula	4	Yes
Casula	Liverpool	4	Yes
Liverpool	Warwick Farm	2	Yes
Warwick Farm	Cabramatta	3	Yes
Cabramatta	Canley Vale	2	Yes
Canley Vale	Fairfield	3	Yes
Fairfield	Yennora	2	Yes
Yennora	Guildford	3	Yes
Guildford	Merrylands	3	Yes
Merrylands	Granville	4	Yes
Merrylands	Harris Park	4	Yes
Central	Mascot	3	Yes
Mascot	Green Square	3	Yes
Green Square	Domestic	3	Yes
Domestic	International	2	Yes
International	Wolli Creek	2	Yes
Central	Domestic	7	Yes
Wolli Creek	Arncliffe	2	Yes
Wolli Creek	Rockdale	4	Yes
Wolli Creek	Kogarah	5	Yes
Wolli Creek	Hurstville	8	Yes
Arncliffe	Banksia	2	Yes
Banksia	Rockdale	2	Yes
Rockdale	Kogarah	2	Yes
Kogarah	Carlton	2	Yes
Kogarah	Hurstville	4	Yes
Carlton	Allawah	2	Yes
Allawah	Hurstville	2	Yes
Hurstville	Penshurst	2	Yes
Hurstville	Sutherland	10	Yes
Hurstville	Mortdale	3	Yes
Penshurst	Mortdale	2	Yes
Mortdale	Oatley	2	Yes
Mortdale	Sutherland	8	Yes
Oatley	Como	4	Yes
Como	Jannali	2	Yes
Jannali	Sutherland	3	Yes
Cronulla	Woolooware	2	Yes
Woolooware	Caringbah	3	Yes
Caringbah	Miranda	3	Yes
Miranda	Gymea	2	Yes

Start	End	Time	Bidirectional
Gymea	Kirrawee	2	Yes
Kirrawee	Sutherland	3	Yes
Miranda	Sutherland	5	Yes
Sutherland	Loftus	2	Yes
Sutherland	Waterfall	10	Yes
Loftus	Engadine	4	Yes
Engadine	Heathcote	3	Yes
Heathcote	Waterfall	5	Yes
Lidcombe	Olympic Park	5	Yes

Vehicle Max Seated	Vehicle Max Standing	Vehicle Cost	Agent Cost Seated	Agent Cost Standing	Agent Cost Waiting	Windup Time	Winddown Time	Final Time	Stop Simulation
960	1680	700	10	15	15	120	120	1200	1260

Vehicle Cost	Agent Cost Seated	Agent Cost Standing	Agent Cost Waiting	Unfinished Penalty
652	10	15	15	150

Route	Modifier	Schedule	
Berowra-Hornsby		Berowra,Mt Ku-Ring Gai,Mount Colah,Asquith,Hornsby	
Hornsby-Gordon		Hornsby,Waitara,Wahroonga,Warrawee,Turramurra,Pymble,Gordon	
Gordon-Chatswood		Gordon,Killara,Lindfield,Roseville,Chatswood	
Gordon-Chatswood	Fast	Gordon,Chatswood	
Lindfield-Chatswood		Lindfield,Roseville,Chatswood	
Chatswood-North Sydney		Chatswood,Atarmon,St Leonards,Wollstonecraft,Waverton,North Sydney	
Chatswood-North Sydney	Fast	Chatswood,St Leonards,North Sydney	
North Sydney-Wynyard		North Sydney,Milsons Point,Wynyard	
Wynyard-Central		Wynyard,Town Hall,Central	
Central-Wynyard	Circle	Central,Museum,St James,Circular Quay,Wynyard	
Bondi Junction-Central		Bondi Junction,Edgecliff,Kings Cross,Martin Place,Town H	St James
Hornsby-Epping		Hornsby,Normanhurst,Thornleigh,Pennant Hills,Beecroft,(	Museum
Epping-Rhodes		Epping,Eastwood,Denistone,West Ryde,Meadowbank,Rhodes	
Epping-Rhodes	Fast	Epping,Eastwood,Rhodes	
Rhodes-Strathfield		Rhodes,Concord West,North Strathfield,Strathfield	
Rhodes-Strathfield	Fast	Rhodes,Strathfield	
Epping-Strathfield	Fast	Epping,Strathfield	
Tallawong-Chatswood		Tallawong,Rouse Hill,Kellyville,Bella Vista,Norwest,Hills Showground,Castle Hill,Cherrybrook,Epping,Macquarie University,Macquarie Park,North Ryde,Chatswood	
Emu Plains-Mount Druitt		Emu Plains,Penrith,Kingswood,Werrington,St Mary's,Mount Druitt	
Mount Druitt-Blacktown		Mount Druitt,Rooty Hill,Doonside,Blacktown	
Mount Druitt-Blacktown	Fast	Mount Druitt,Blacktown	
Richmond-Schofields		Richmond,Claredon,Windsor,Mulgrave,Vineyard,Riverstone,Schofields	
Schofields-Blacktown		Schofields,Quakers Hill,Marayong,Blacktown	
Blacktown-Parramatta		Blacktown,Seven Hills,Toongabbie,Pendle Hill,Wentworthville,Westmead,Parramatta	
Blacktown-Parramatta	Fast	Blacktown,Parramatta	
Westmead-Parramatta		Westmead,Parramatta	
Parramatta-Granville		Parramatta,Harris Park,Granville	
Parramatta-Granville	Fast	Parramatta,Granville	
Parramatta-Strathfield	Fast	Parramatta,Strathfield	
Parramatta-Strathfield	Semi-Fast	Parramatta,Granville,Auburn,Lidcombe,Strathfield	
Granville-Auburn		Granville,Clyde,Auburn	
Granville-Auburn	Fast	Granville,Auburn	
Auburn-Lidcombe		Auburn,Lidcombe	
Lidcombe-Strathfield		Lidcombe,Flemington,Homebush,Strathfield	
Lidcombe-Strathfield	Fast	Lidcombe,Strathfield	
Flemington-Strathfield		Flemington,Homebush,Strathfield	
Strathfield-Central	Super Fast	Strathfield,Central	
Strathfield-Central	Fast	Strathfield,Redfern,Central	
Strathfield-Central	Fast2	Strathfield,Burwood,Redfern,Central	
Strathfield-Central	Semi-Fast	Strathfield,Burwood,Ashfield,Newtown,Redfern,Central	
Strathfield-Central		Strathfield,Burwood,Croydon,Ashfield,Summer Hill,Lewisham,Petersham,Stanmore,Newtown,Macdonaldtown,Redfern,Central	
Sydenham-Central		Sydenham,St Peters,Erskineville,Redfern,Central	
Sydenham-Central	Fast	Sydenham,Redfern,Central	
Wolli Creek-Central	Fast	Wolli Creek,Redfern,Central	
Wolli Creek-Sydenham		Wolli Creek,Tempe,Sydenham	
Wolli Creek-Sydenham	Fast	Wolli Creek,Sydenham	
Wolli Creek-Central	Airport-Fast	Wolli Creek,International,Domestic,Central	
Wolli Creek-Central	Airport	Wolli Creek,International,Domestic,Green Square,Mascot,Central	
Kogarah-Wolli Creek	Fast	Kogarah,Wolli Creek	
Kogarah-Wolli Creek	Semi-Fast	Kogarah,Rockdale,Wolli Creek	
Kogarah-Wolli Creek		Kogarah,Rockdale,Banksia,Arncliffe,Wolli Creek	
Hurstville-Kogarah	Fast	Hurstville,Kogarah	
Hurstville-Kogarah		Hurstville,Allawah,Carlton,Kogarah	
Mortdale-Hurstville		Mortdale,Penshurst,Hurstville	
Sutherland-Hurstville		Sutherland,Jannali,Como,Oatley,Mortdale,Penshurst,Hurstville	
Sutherland-Hurstville	Fast	Sutherland,Hurstville	
Sutherland-Hurstville	Semi-Fast	Sutherland,Jannali,Como,Oatley,Mortdale,Hurstville	
Cronulla-Sutherland		Cronulla,Woolooware,Caringbah,Miranda,Gymea,Kirrawee,Sutherland	
Waterfall-Sutherland		Waterfall,Heathcote,Engadine,Loftus,Sutherland	
Campsie-Sydenham		Campsie,Canterbury,Hurlstone Park,Dulwich Hill,Marrickville,Sydenham	
Campsie-Sydenham	Semi-Fast	Campsie,Marrickville,Sydenham	
Campsie-Sydenham	Fast	Campsie,Sydenham	
Bankstown-Campsie		Bankstown,Punchbowl,Wiley Park,Lakemba,Belmore,Campsie	
Bankstown-Campsie	Fast	Bankstown,Campsie	
Sefton-Bankstown		Sefton,Birrong,Yagoona,Bankstown	
Sefton-Lidcombe		Sefton,Regents Park,Berala,Lidcombe	
Sefton-Lidcombe	Fast	Sefton-Lidcombe	
Cabramatta-Sefton		Cabramatta,Carramar,Villawood,Leightonfield,Chester Hill,Sefton	
Cabramatta-Sefton	Fast	Cabramatta,Sefton	
Merrylands-Granville		Merrylands,Granville	
Merrylands-Parramatta		Merrylands,Harris Park,Parramatta	
Cabramatta-Merrylands		Cabramatta,Canley Vale,Fairfield,Yennora,Guildford,Merrylands	
Liverpool-Cabramatta		Liverpool,Warwick Farm,Cabramatta	
Glenfield-Liverpool		Glenfield,Casula,Liverpool	
Leppington-Glenfield		Leppington,Edmondson Park,Glenfield	
Macarthur-Glenfield		Macarthur,Campbelltown,Leumeah,Minto,Ingleburn,Macquarie Fields,Glenfield	
Glenfield-Revesby	Fast	Glenfield,Revesby	
Glenfield-Revesby		Glenfield,Holsworthy,East Hills,Panania,Revesby	
Revesby-Kingsgrove	Fast	Revesby,Kingsgrove	
Revesby-Kingsgrove		Revesby,Padstow,Riverwood,Narwee,Beverly Hills,Kingsgrove	
Kingsgrove-Wolli Creek	Fast	Kingsgrove,Wolli Creek	
Kingsgrove-Wolli Creek		Kingsgrove,Bexley North,Bardwell Park,Turrella,Wolli Creek	
Glenfield-Wolli Creek	Semi-Fast	Glenfield,Revesby,Kingsgrove,Wolli Creek	
Lidcombe-Olympic Park		Lidcombe,Olympic Park	

Name	Gap	Offset	Finish	Schedule Segments
Berowra-Central	30	0	1,200	Berowra-Hornsby,Hornsby-Gordon,Gordon-Chatswood,Chatswood-North Sydney,North Sydney-Wynyard,Wynyard-Central
Central-Berowra	30	0	1,200	Central-Wynyard,Wynyard-North Sydney,North Sydney-Chatswood,Chatswood-Gordon,Gordon-Hornsby,Hornsby-Berowra
Hornsby-Central via Chatswood	30	0	1,200	Hornsby-Gordon,Gordon-Chatswood,Chatswood-North Sydney,North Sydney-Wynyard,Wynyard-Central
Central-Hornsby via Chatswood	30	0	1,200	Central-Wynyard,Wynyard-North Sydney,North Sydney-Chatswood,Chatswood-Gordon,Gordon-Hornsby
Gordon-Central	15	0	1,200	Gordon-Chatswood,Chatswood-North Sydney,North Sydney-Wynyard,Wynyard-Central
Central-Gordon	15	0	1,200	Central-Wynyard,Wynyard-North Sydney,North Sydney-Chatswood,Chatswood-Gordon
Hornsby-Central via Epping	15	0	1,200	Hornsby-Epping,Epping-Rhodes,Rhodes-Strathfield,Strathfield-Central Fast
Central-Hornsby via Epping	15	0	1,200	Central-Strathfield Fast,Strathfield-Rhodes,Rhodes-Epping,Epping-Hornsby
Epping-Central	15	0	1,200	Epping-Rhodes,Rhodes-Strathfield,Strathfield-Central Fast
Central-Epping	15	0	1,200	Central-Strathfield Fast,Strathfield-Rhodes,Rhodes-Epping
Tallawong-Chatswood	10	0	1,200	Tallawong-Chatswood
Chatswood-Tallawong	10	0	1,200	Chatswood-Tallawong
Bondi Junction-Central	10	0	1,200	Bondi Junction-Central
Central-Bondi Junction	10	0	1,200	Central-Bondi Junction
Emu Plains-Central	15	0	1,200	Emu Plains-Mount Druitt,Mount Druitt-Blacktown,Blacktown-Parramatta Fast,Parramatta-Strathfield Fast,Strathfield-Central Fast
Central-Emu Plains	15	0	1,200	Central-Strathfield Fast,Strathfield-Parramatta Fast,Parramatta-Blacktown Fast,Blacktown-Mount Druitt,Mount Druitt-Emu Plains
Richmond-Central	30	0	1,200	Richmond-Schofields,Schofields-Blacktown,Blacktown-Parramatta Fast,Parramatta-Strathfield Fast,Strathfield-Central Fast
Central-Richmond	30	0	1,200	Central-Strathfield Fast,Strathfield-Parramatta Fast,Parramatta-Blacktown Fast,Blacktown-Schofields,Schofields-Richmond
Schofields-Central	30	0	1,200	Schofields-Blacktown,Blacktown-Parramatta Fast,Parramatta-Strathfield Semi-Fast,Strathfield-Central Fast
Central-Schofields	30	0	1,200	Central-Strathfield Fast,Strathfield-Parramatta Semi-Fast,Parramatta-Blacktown Fast,Blacktown-Schofields
Blacktown-Central	15	0	1,200	Blacktown-Parramatta,Parramatta-Strathfield Semi-Fast,Strathfield-Central Fast
Central-Blacktown	15	0	1,200	Central-Strathfield Fast,Strathfield-Parramatta Semi-Fast,Parramatta-Blacktown
Parramatta-Central	15	0	1,200	Parramatta-Granville,Granville-Auburn,Auburn-Lidcombe,Lidcombe-Strathfield,Strathfield-Central Semi-Fast
Central-Parramatta	15	0	1,200	Central-Strathfield Semi-Fast,Strathfield-Lidcombe,Lidcombe-Auburn,Auburn-Granville,Granville-Parramatta
Flemington-Central	15	0	1,200	Flemington-Strathfield,Strathfield-Central
Central-Flemington	15	0	1,200	Central-Strathfield,Strathfield-Flemington
Macarthur-Central	15	0	1,200	Macarthur-Glenfield,Glenfield-Revesby,Revesby-Kingsgrove Fast,Kingsgrove-Wolli Creek Fast,Wolli Creek-Central Airport
Central-Macarthur	15	0	1,200	Central-Wolli Creek Airport,Wolli Creek-Kingsgrove Fast,Kingsgrove-Revesby Fast,Revesby-Glenfield,Glenfield-Macarthur
Revesby-Central	15	0	1,200	Revesby-Kingsgrove,Kingsgrove-Wolli Creek,Wolli Creek-Central Airport
Central-Revesby	15	0	1,200	Central-Wolli Creek Airport,Wolli Creek-Kingsgrove,Kingsgrove-Revesby
Central-Cronulla	15	0	1,200	Central-Sydenham Fast,Sydenham-Wolli Creek Fast,Wolli Creek-Kogarah Semi-Fast,Kogarah-Hurstville Fast,Hurstville-Sutherland Fast,Sutherland-Cronulla
Cronulla-Central	15	0	1,200	Cronulla-Sutherland,Sutherland-Hurstville Fast,Hurstville-Kogarah Fast,Kogarah-Wolli Creek Semi-Fast,Wolli Creek-Sydenham Fast,Sydenham-Central Fast
Central-Waterfall	30	0	1,200	Central-Sydenham Fast,Sydenham-Wolli Creek Fast,Wolli Creek-Kogarah Semi-Fast,Kogarah-Hurstville Fast,Hurstville-Sutherland Semi-Fast,Sutherland-Waterfall
Waterfall-Central	30	0	1,200	Waterfall-Sutherland,Sutherland-Hurstville Semi-Fast,Hurstville-Kogarah Fast,Kogarah-Wolli Creek Semi-Fast,Wolli Creek-Sydenham Fast,Sydenham-Central Fast
Central-Sutherland	30	0	1,200	Central-Sydenham Fast,Sydenham-Wolli Creek Fast,Wolli Creek-Kogarah Semi-Fast,Kogarah-Hurstville Fast,Hurstville-Sutherland Semi-Fast
Sutherland-Central	30	0	1,200	Sutherland-Hurstville Semi-Fast,Hurstville-Kogarah Fast,Kogarah-Wolli Creek Semi-Fast,Wolli Creek-Sydenham Fast,Sydenham-Central Fast
Central-Mortdale	15	0	1,200	Central-Sydenham,Sydenham-Wolli Creek,Wolli Creek-Kogarah,Kogarah-Hurstville,Hurstville-Mortdale
Mortdale-Central	15	0	1,200	Mortdale-Hurstville,Hurstville-Kogarah,Kogarah-Wolli Creek,Wolli Creek-Sydenham,Sydenham-Central
Central-Sefton	30	0	1,200	Sefton-Bankstown,Bankstown-Campsie,Campsie-Sydenham Semi-Fast,Sydenham-Central Fast
Sefton-Central	30	0	1,200	Central-Sydenham Fast,Sydenham-Campsie Semi-Fast,Campsie-Bankstown,Bankstown-Sefton
Central-Bankstown	30	0	1,200	Bankstown-Campsie,Campsie-Sydenham Semi-Fast,Sydenham-Central Fast
Bankstown-Central	30	0	1,200	Central-Sydenham Fast,Sydenham-Campsie Semi-Fast,Campsie-Bankstown
Central-Campsie	15	0	1,200	Campsie-Sydenham,Sydenham-Central
Campsie-Central	15	0	1,200	Central-Sydenham,Sydenham-Campsie
Leppington-Parramatta	30	0	1,200	Leppington-Glenfield,Glenfield-Liverpool,Liverpool-Cabramatta,Cabramatta-Merrylands,Merrylands-Parramatta
Parramatta-Leppington	30	0	1,200	Parramatta-Merrylands,Merrylands-Cabramatta,Cabramatta-Liverpool,Liverpool-Glenfield,Glenfield-Leppington
Liverpool-Parramatta	30	0	1,200	Liverpool-Cabramatta,Cabramatta-Merrylands,Merrylands-Parramatta
Parramatta-Liverpool	30	0	1,200	Parramatta-Merrylands,Merrylands-Cabramatta,Cabramatta-Liverpool
Central-Liverpool	15	0	1,200	Central-Strathfield Fast,Strathfield-Lidcombe Fast,Lidcombe-Sefton,Sefton-Cabramatta,Cabramatta-Liverpool
Liverpool-Central	15	0	1,200	Liverpool-Cabramatta,Cabramatta-Sefton,Sefton-Lidcombe,Lidcombe-Strathfield Fast,Strathfield-Central Fast
Cabramatta-Central	15	0	1,200	Cabramatta-Merrylands,Merrylands-Granville,Granville-Auburn Fast,Auburn-Lidcombe,Lidcombe-Strathfield Fast,Strathfield-Central Fast
Central-Cabramatta	15	0	1,200	Central-Strathfield Fast,Strathfield-Lidcombe Fast,Lidcombe-Auburn,Auburn-Granville Fast,Granville-Merrylands,Merrylands-Cabramatta
North Sydney-Central	5	0	1,200	North Sydney-Wynyard,Wynyard-Central
Central-North Sydney	5	0	1,200	Central-Wynyard,Wynyard-North Sydney
City Circle Clockwise	5	0	1,200	Central-Wynyard,Wynyard-Central Circle
City Circle Counter	5	0	1,200	Central-Wynyard Circle,Wynyard-Central
Lidcombe-Olympic Park	15	0	1,200	Lidcombe-Olympic Park
Olympic Park-Lidcombe	15	7	1,200	Olympic Park-Lidcombe

Traffic Multiplier
0
0.025
0.05
0.05
0.05
0.05
0.05
0.05
0.05
0.05
0.05
0.05
0.05
0.05
0.05
0.05
0.05
0.05
0.05
0.05
0.025
0

Traffic Multiplier
0
0.015
0.041
0.085
0.141
0.071
0.036
0.033
0.035
0.037
0.044
0.068
0.081
0.139
0.092
0.043
0.027
0.023
0.017
0.008
0.003
0