Network Traffic Simulator: Designing an Extensible Traffic Management System Using Python

Julia Ruiter

Yannis Velegrakis, Advisor

Ioana Karnstedt-Hulpus, Reader

MSc Applied Data Science

Department of Natural Sciences

July 11, 2022

Abstract

This paper documents the creation of an extensible traffic management system that can be used to simulate various graph problems from congestion in city traffic, to distribution and shipping logistics, to internet traffic. The Network Traffic Simulator design process and motivation have been fully described, and the paper is complete with a tutorial on how to use the software to solve your own network traffic problems.

Acknowledgments

I'm extremely grateful for my partner, Warren Fletcher, for all the support he has provided me (both professionally and emotionally) in the duration of both thesis and this entire master's program. He has given invaluable guidance in writing and designing a proper/professional piece of software, and has patiently helped fill in the gaps in my computer science knowledge, pointing me in the right direction for data structure and algorithm usage. This project would not be possible without the many design debates and ensuing suggestions from him.

Contents

List of Figures

Chapter 1

Introduction

Making your morning commute to the office by car, waiting for your online purchase to arrive days or weeks after ordering, using the Tor browser to access articles or sites not available in your country; these are systems of describing the movement of people, things, or information from point A to point B, with various levels of intervention or autonomy in the process. Each of these situations isn't an action in isolation, but part of a larger system of units moving over an underlying infrastructure. The purpose of this project is to create a framework that can model each of these systems and more.

This generality is at odds with existing popular software for simulation modeling, which typically specialize in one particular network type to simulate (ex: urban car traffic) [LWB+18]. Simulations built using these programs have a wealth of domain-specific knowledge and available parameters to tune. However, this can become troublesome when a user wants to extend the package to other network types. Due to domain-specific attribute dependencies, users must be clever about how they use and overlap various domain-specific packages to model other system types. Generally, this results in adding the features of one to the other rather than modelling something new (ex: adding NS2 to SUMO to model telecommunication protocols or mixed-mode travel) [LC08] [SKMR14]. For a more detailed look at existing programs and related research, see chapter 2, *"Existing Traffic Simulation Models"*.

The new Traffic Management system/simulation framework outlined in this paper aims to provide an alternative to this by being explicitly designed around the question: *How can one natively model the idiosyncrasies of a system without loss of generality in the underlying software?* A detailed discussion on why particular design choices were made on the microscopic and macroscopic

level can be found in chapter 3, *"Design Motivations"*, but the general guiding considerations used for ensuring software generality and adaptability to various use-cases are:

- **Modular**: each component of the system is distinct and isolated. Though an element may be dependent on another class, any functions pertaining to it's behavior are passed back down to the class it affects before updating. This allows for further layers of abstraction or separation to be added (or removed) if necessary.
- **Abstract**: if a system can be designed such that a network simulation as a concept can be modeled, then the system should work for all subclasses of network simulation. This means that a simulation should be possible on the most bare-bones version of a network/object system.
- **Consistent**: subclasses have been ignored in this version of the software and should continue to be absent as much as possible. Allowing for subclasses allows for use-case-specific functions to creeps into the code; by forcing functions to be general, you ensure the program is adaptable to use cases not yet considered.
- **Attribute agnostic**: as much as possible, object attributes should be non-essential; this follows from "Abstract". Though a "default" configuration value has been set for essential variables, nearly all of the default values are set to the least-restrictive values possible.
- **Accuracy over performance**: though many calculations could be simplified by assuming fixed parameters, real-world systems are seldom consistent in practice. By allowing each individual component the ability to have unique attributes, the user can build more complex and nuanced simulations.

As the software project this thesis refers to is open-source and still a work in progress, some of the details in chapter 5, *"Using the Network Traffic Simulator"*, may become outdated as the project grows and evolves. Anticipated additions in future versions and an outline for creating a visualizer and UI interface can be found in chapter 6, *"Next Steps"*. The current version and place where user suggestions and contributions for additional features and implementations can be made can be found at:

`https://github.com/julialruiter/Traffic_Simulator`
`v1.0.0: fb860d1`

Chapter 2

Existing Traffic Simulation Models

The majority of popular open-source network modeling software focus explicitly on a single network type. The two most widely used tools and models are SUMO, which specializes in tackling urban traffic simulations, and NS2, which specializes in communication networks. //

SUMO has become the de-facto simulator used to research mobility problems (IEEE eXplore notes nearly 500 citations for the program's inaugural conference paper), allowing researchers to plan and execute urban planning schemes that would be too costly (or time consuming, or flat-out impossible) to enact [LWB+18]. SUMO works by giving its users a graphical user interface where they can import geospatial maps data (typically from Open Street Maps, but users can also draw a graph) and overlay traffic events. In the GUI, users can add lanes to existing roads or change the type of vehicles allowed per lane (car, bus, high-occupancy vehicle, bike, etc), adding edges to the network and adding (or removing) restrictions to those edges [D+22]. While heavy emphasis is placed on using the GUI, users can also opt to import files and run simulations using Python.

Once the road network setup is complete, the user can move on to the "Demand" setup, which is how SUMO describes adding vehicular traffic. Through the GUI (or importing xml files), users can add vehicles, specifying numerous vehicle attributes like vehicle type and route information[D+22]. The car creates a *demand* on the network to enter and exit at a particular timestamp, then the network simulation tries to execute it.

While these features are what make SUMO excel at urban mobility simulation, because of its dependence on geographic location, its use cases cannot be abstracted out [LWB+18]. This means that even similar network problems

like pedestrian mobility on the same network, simulating truckers who communicate with one another on the same network, or allowing an individual to move from one type of transportation to another require the additional use-case-specific packages and simulators to run on top [NBKL21], [LWB+18], [SKMR14]. The intersection of traffic and mobility (as in the aforementioned adaptations to SUMO) has resulted in an entire genre of simulators called VANET (Vehicular Ad-Hoc Network). However, VANET simulations cannot be evaluated directly by SUMO nor the appended packages, thus requiring yet another set of simulation packages to run on top of SUMO [LC08].

Of course, the same network traffic systems outlined above can be modeled outside of the SUMO environment entirely. Far preceding the creation of SUMO, papers on trucking and transport simulations can be found dating back to the '90s, even using programs like Microsoft Excel to run them [Dag94]. This leads us to question whether SUMO's painstakingly created attributes and dependencies are even necessary for spinoff simulations.

To see an example of a more generalized framework in action, we can look at the other popular network traffic simulation framework: NS2. NS2 is a "object oriented simulator" tool which was designed explicitly for the simulation and assessment of computer communication networks. The user manual "Introduction to Network Simulator NS2" details how to use the system for an extensive variety of communication-based simulations it supports have grown extensively in the 30+ years of its existence [IH11]. Because it is a solid base for simulations where objects may talk with one another, it has seen use beyond its original scope (including many uses in conjunction with SUMO [LWB+18]).

While SUMO allows users to intuitively approach simulation from a visual perspective, NS2 does not, nor should it: computer communication is somewhat abstract and very intangible, so users to not expect a GUI for setup (though users can then run the set up simulation in a viewing window). Instead NS2 presents itself as a library users can install and use with Python and runs on C++ objects. NS2 further differs in that is enables the use of distributed computing to run the traffic events [IH11]. Rather than modeling location of an object on an edge directly, NS2 operates by calculating the time delay between nodes, and extrapolating location based on that. This corroborates NS2's claim to be a purely mathematical model [IH11].

The design of this traffic simulator has drawn heavily on SUMO for guidance in accommodating "vehicle" attributes; this allows for nuanced

simulations to run and supports full customizaibility from the user in how and when the "vehicle" objects can make changes to their route. But diverging from SUMO and borrowing from NS2, this traffic simulator has tried to keep the simulation as mathematical and modular as possible, ensuring that the components can run in any order and regardless of user interference. Taking inspiration from the detail of SUMO and the flexibility of NS2, this project aims to create a self-contained simulation environment that can handle a range of use-cases (provided the user has adequate configuration data available).

Chapter 3

Design Motivations

The goal of this project was to design and code a general solution to the traffic modeling problem. After all–if one can prove that the generalized solution works, then any special case should, too, thus satisfying the aim to create an extensible and multi-use simulation model.

To prove so requires combining graph theory, combinatorics, and data structures. However, computers require a bit more work to conform to the natural mathematics of graph theory as they force discreteness. This meant that special considerations had to be made to emulate the continuous real-world systems it may be required to simulate. With real-world precision and accuracy as the utmost goal, the system was designed with the following criteria:

3.1 Network Structure: how should the network be stored in memory?

Nodes are connection points for several edges, like road intersections, cellular towers, etc. By utilizing graph data structures, one can also take advantage of the numerous existing path-finding algorithms and easily keep track of connected pair of nodes (and their directionality).

The advantages of designing a directed graph model rather than undirected are obvious: car lanes force traffic in a particular direction, and packets of information cannot necessarily be transmitted in both directions (especially if there are intermediary steps, like encryption, that are not reversible).

Further continuing with intuitive design decision, it followed that the network needed to be directional, and that multiple parallel edges between a pair of nodes should be allowed. Directionality's reason is easy to sport:

Directionality has another advantage in that it allows for multiple parallel edges. This has the added benefit of intuitively and adaptably being able to increase (or restrict) edge capacity, serving as another parameter the user can tune to mimic real-world constraints. This ensures that this traffic simulator has the flexibility to model microscopic and macroscopic traffic trends [LWB+18]. Using normal commuting traffic as an example, a user can test whether or not adding another lane to a freeway (or adding lane accessible to cars only of a certain type) would ease congestion as they are able to view metrics and positionality per timestamp of the entire system (macroscopic), the set of edges between two nodes or each individual node (mesoscopic), and the individual cars on those roads themselves (microscopic) [LWB+18].

Though some traffic simulations in existence use adjacency matrices to define neighbouring nodes [GPK02], this is not a scalable, nor desirable, solution . While adjacency matrices are intuitive for humans to read and understand, since most networks are fairly sparse they end up storing a lot of NULL values and require n! complexity to process those NULLs. While little heed is paid to memory conservation in the rest of this simulation's design process, the idea of holding space for nonexistent edges did not seem like a very good idea. Instead, it makes sense to think more about the interactions of node-edge pairs via their object pointers. Adjacency matrices acknowledge that an edge is defined by its originating and terminal nodes, but fails to demonstrate how a node is only interesting because of the set of inbound and outbound edges it connects. To capture both of these ideas, dictionary mappings were utilized to identify and link nodes to edges and edges to nodes, allowing either or both interactions to be utilized, depending whichever one was more intuitive for the particular action being done at any point in the simulation process.

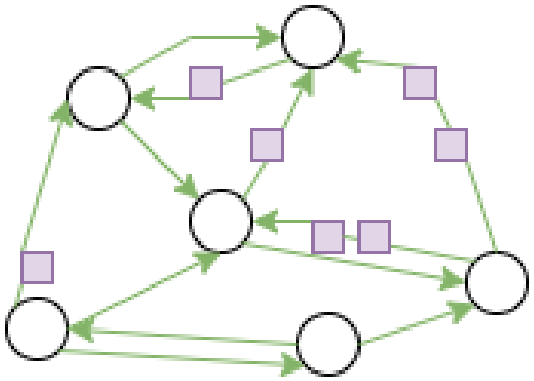

Figure 3.1 Proposed structure: random directional network with objects on it, with one edge per possible object-carrying "lane" (non-redundant)

3.2 How does the traffic network change state, processing car movement?

A methodical process is needed to execute all the actions that can take place on each tick of the simulator (tick being the function that executes said changes for each incremental timestamp in the simulation runtime). Once all available or possible actions have been made for that tick, the simulation is considered to be in a new state, and that information about that state is returned to the user.

The network object consists of nodes and edges, so it makes sense to break down the action queue to these levels as well. However, you can't just arbitrarily run all nodes and edges as there is an inherent order and hierarchy to them. As edges are defined by their start and end nodes, it makes sense to have edge tick processes as a dependent of node tick processes. Furthermore, because nodes are capable of having multiple inbound and outbound edges, it is essential to process a particular node's edge ticks together to ensure as smooth and realistic of a movement between edges as possible.

To ensure that each component of a network is only processed once, on each tick, the network tick function iterates through all nodes listed in the network's node dictionary. Each node, in turn, processes each outbound edge in its adjacent outbound edges dictionary. The network tick and its subsequent node and edge ticks will be performed as many times as necessary until the cars run out of energy (more on that later in the **Movement** section) *or* no

more advances can be made with the current remaining energy potentials, and report out the number of loops needed to do so and the total percent of potential energy used. These values can be used as a proxy for network congestion, though more as a benchmark between ticks or simulations than as an absolute value of ability.

To ensure that no node nor edge is favored (ei: always has their candidate cars move before any other node or edge's), the order in which these items are iterated is shuffled with each pass. While this helps immensely with simulation accuracy, the random nature is what prevents the output numbers from being a reliable and accurate metric of network (in)action.

3.3 Where should car location be stored?

A traffic simulation is not very insightful without considering the things which themselves cause traffic–Who, or what, keeps track of where the "cars" are? Though innocuous, the question leads to some philosophical fancies that need to be addressed before determining who (or what) is in charge of your position.

If you're driving home at midnight, you might choose to take a faster route or a more scenic route, you might pull over to look at the citylights or take a pause because you're feeling sleepy, or you might drive a bit over the speed limit up because there's not likely to be any cops on the road at this hour of night. It feels like you own the road and you have full control over where you are right here right now.

But what if you (are trying to) drive home 5 o'clock Friday in the thick of traffic? Yes, you *chose* to take a particular route home, but now you're stuck in bumper-to-bumper traffic and can't move forward til the car in front of you decides to (or can at all). While you may be in control of your car, you don't have full control over your ability to move, and thus over your position. This leads to the inevitable conclusion that a simulation will be most accurate if the network controls the cars rather than letting the cars control themselves, meaning that current position must be stored on the objects over which the cars are moving: the edges.

3.4 Discrete versus continuous systems: how is position represented?

Since edges store car locations and move them along a particular distance each unit of time, it's tempting to think of positions in terms of capacity. If an edge is 10 units long and cars travel 2 units per time, there are 5 positions a car can be, so a car's position can be stored as its index in the capacity queue. This makes movements easy to simulate and requires a minimal, fixed storage space no matter the amount of cars on the network. So this is that design choice, right?

...well, not quite. Sure, the example above is easy to quantify, but breaks down when situations arise that obstruct predictable movement, ranging from traffic jams to even just switching to an edge with a different speed limit. If a car is unable to move its full potential, it's then unable to move into the next discrete state. So where does it go? And where do any cars after it go?

The problem with defining position by the distance a car can go at it's maximum speed is that there's a lot of unaccounted distance. If a car is going 10 m/min, then the position would be defined as 10 meters later (if the unit time distance of the simulation were in minutes)–in a road stretch of 60 meters, there would be 6 possible positions.

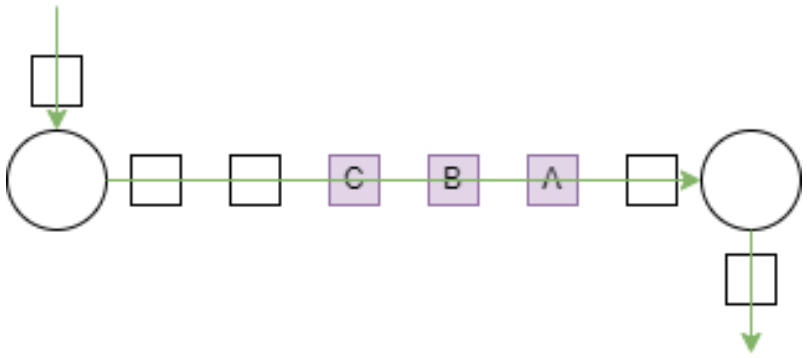

Figure 3.2 Example discrete system with 6 slots, 3 of which currently occupied

Now let's say that the car (car A) has reached the end of the road segment and is trying to turn onto a crossroad, but the crossroad is completely full. Car A must stop at the end of the road segment and wait for an opening.

The car behind (car B), meanwhile, is still driving 10 m/min but must stop before crashing into the halted car in front, but any cars behind it are still capable of moving fully into the next position in the queue. Where is Car B? It cannot be in the final position (as Car A is still occupying it), and the penultimate position is now taken up by Car C which was one position behind Car B.

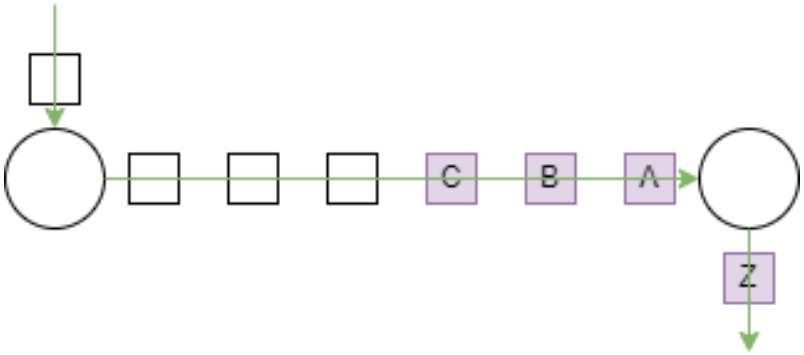

Figure 3.3 No obstructions yet: all cars have proceeded one slot forward

Faced with this situation in a discrete system, we must compromise on simulation accuracy: either you allow cars to pile up on positions (defeating the purpose of a queue *and* unrealistically depicting car location), or you prevent cars from moving at all (artificially causing traffic on the current edge when there is, in fact, room to move. This cascades onto other connected edges and may stall the whole simulation). While option one doesn't seem too terrible, it runs the risk of allowing cars to "jump" over halted cars; if car C's destination is the location car A is stuck, then when car C piles onto the position two timestamps later, it registers as completing its journey despite that being impossible in a real-world scenario.

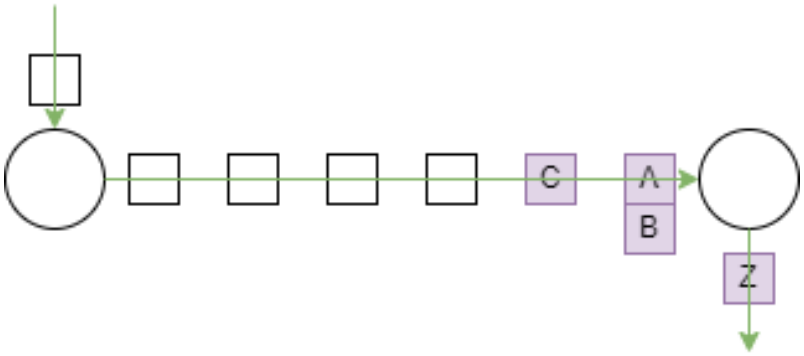

Figure 3.4 Car A is stuck, car B and C advance, causing car B to share a spot with car A

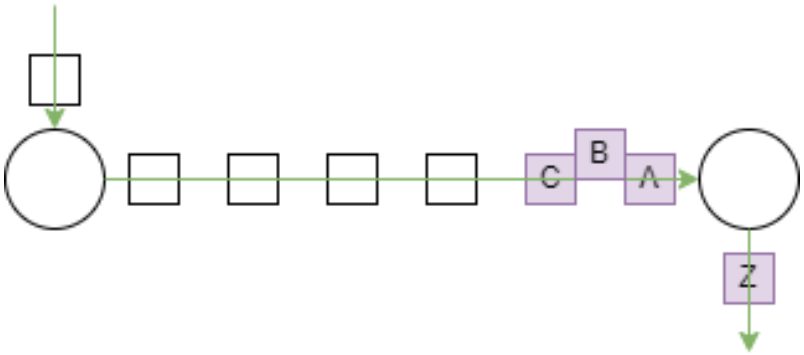

Figure 3.5 Car A is stuck, car B advances to an in-between stage, Car C advances one slot forward

It is clear that some other solution is needed if a realistic simulation is to be created. And that solution is simple: switch to floating point positions. By mapping cars to their exact position from 0 to maximum edge length, the user can be explicit about how long each car is and how close the cars are allowed to be to each other, all the while ensuring accuracy in car behavior by preventing skipping. While this adds additional complexity to movement calculations, the resulting affinity for precision solves any practical or logical issues that a discrete system would incur.

3.5 On Paths, Origins, and Destinations

"How do I get from Point A to Point B?" and what even constitutes a "point"? Path-finding algorithms typically find the fastest/shortest/"best" route be-

tween any 2 given nodes, which would imply that nodes are the points. This intuitively makes sense and works great for simulation paths with bounded, real-world constraints. Take (a very simplified example of) email communication: emails will always start at one machine (node), travel to a server (node), and perhaps be passed onto another server for the receiver to view (node); there might be more or less steps between each landmark, but the only place a message can originate is at one of the endpoints.

However, cars do not spawn in the middle of a four-way intersection, nor does it make sense to create a network model where every possible parallel parking spot is a separate node in a network. For one, someone could do a terrible parking job and take up two spots, therefore creating some new start or end node somewhere between the existing spots. But the more pressing issue with breaking a road into n continuous, sequential, connected segments is the same sort of unwanted inefficiency as the adjacency matrices proposed earlier: there's a lot space and computation wasted on pairs that generally provide no function to the simulation.

This led to design decision to turn the usual graph structure on its head, and instead allow cars to enter and leave the network from the edges themselves. To make this work for traditional node-to-node paths, the car placement mechanism has been written in such a way that if no specific edge location has been specified for a start and end point, a path is selected based on those nodes and the "car" is placed at position 0 along the first edge (and finishes its journey at the maximum length of the final edge, or effectively at the terminal node).

Since start and end locations have been set to edges, path finding calculations also are done on an edge to edge basis. As edges only know their bounding nodes (and nodes their adjacent edges), calculations are done on the network level, allowing chaining between edge dictionaries of nodes and node dictionaries of edges to build potential paths.

3.6 How do cars move along their path?

A simulation is practically useless unless it models the movement of objects predictably and (semi-)realistically; the problem of how to model node-crossing caused quite some consternation.

For an intersection consisting of one inbound edge and one outbound edge, the logic is simple: when a car reaches the end of one edge, check the following edge and place it there if there is room. Even in the simple scenario, there are certain complexities that arise when translating that from math theory to computer code, and compound when adding more degrees of freedom by adding more inbound and outbound edges:

- What counts as "room"?
- How far does a car go on to the following edge?
- What if the two edges have different speeds?
- What if cars from two (or more!) different inbound edges want to move onto the same new edge?
- Are cars allowed to change their path?

And this list doesn't even consider what happens what other optional attributes like stoplights are added into the mix!

"Room" is the easiest to answer and was already hinted at a few sections ago: a car will move as far as it possibly can, given the internal and external constraints on its movement. This holds for movement within a given edge and across edges (node-crossing).

In describing why a discrete system didn't work, the word "potential" was used to describe a car's possible range of movement; this was not on accident. Even on a floating-point system, the maximum distance a car is allowed to go in one unit of time (tick) is an essential calculation for determining all aspects of a car's movement. This value is denoted as "maximum_tick_potential", which was derived from leaning into the field of physics:

- "Work" is defined as the amount of energy expended to move a certain distance in a certain period of time.
- An object at rest has some arbitrary value of potential energy.
- The law of conservation of energy states that the total energy of a system must remain constant.

It follows that the total energy of the universe remains constant, so any energy expended on moving an object is subtracted from the object's potential energy to ensure the balance remains. By defining a car's "potential" for movement during one tick, we can define its actual moment as a proportion of that, allowing multiple actions to be done in one tick (as long as there is energy potential to spend). Furthermore, we can calculate the total un-expended energy at the end of the tick of the entire network *or* for a particular edge and use this as a metric for how backed-up or congested the network is.

3.6.1 Car movement using tick potential

The following steps describe how a car can possibly move within one tick.

1. Each car starts with it's maximum potential energy at the start of a tick (default = 1). This is the currently available potential energy.

2. Any car that takes an action on the list below is done moving for the current tick and will be moved from the current_cars queue to a processed_cars queue.

3. If a car has been added to the simulation and is waiting to enter the network, check if its start position is open, and add it to the edge at the location if possible (or keep it in the waiting queue if not). Placement on the edge uses up the full energy potential to prevent cars that enter late in the network tick loop from moving more than is realistic.

4. Evaluate how far a car can possibly move along its current edge: The maximum possible distance is the edge's speed limit times the currently available potential energy, but the actual possible travel distance is that to any obstacle in front of it. If there is a car in front, the car can only move to behind it; if the edge ends, the car can only go to the end at the current speed; if the car reaches its path end position, it leaves the network entirely; otherwise, the car can go its full potential.

5. Calculate the work done to get to that position: divide distance travelled by maximum potential distance (speed limit).

6. Subtract work done from the remaining available energy (or maximum potential energy). If there is still energy left and a car has reached the end of the current edge, proceed. Otherwise, wait til the next tick.

7. Evaluate if the car can proceed on to the next edge: nodes might have a (time) penalty for crossing (such as time to physically cross an intersection); if the car does not have enough energy left after "paying" this penalty, then the car cannot proceed further and must wait til the next tick.

8. Select the next edge in the path: for some cars, this is simply the next edge in the path list; other cars (depending on car type) may require a calculation to choose a new path first.

9. Repeat steps 3 through 8 as long as there is energy remaining *or* until the car is forced to wait.

3.6.2 Multiple Inbound Edges: Mitigating preferential treatment of nodes or edges

The issue of multiple cars across several edges eligible to change onto the same new edge is solved, in part, by the random shuffling of each node tick and edge tick order. Some randomness will persist as the order edges are processed may affect whether other cars are even eligible to enter after it, but one could argue that the randomness accurately portrays real-world indecisiveness (at least for car traffic networks) and thus is not something to fear, but rather embrace. However, if this is deemed undesirable by the user, they can mitigate the effects by choosing a small enough tick time that differences are negligible. Tick time can be adjusted by adjusting the scalar values for the maximum speed parameter of the edges.

3.7 What kinds of object attributes should be required for the simulation to run?

Since the simulation software should allow for full flexibility in what types of network systems it models, it was important to make sure the simulation requires as few mandatory fields as possible to produce a reasonable simulation, but allow for additional parameters inherent to a particular system.

Following basic graph theory, the essential attributes for the network (nodes and edges) itself is only what is strictly required to make a graph: a unique identifier per object, and for edges a value to link each end to it respective node. However, a slew of additional attributes (like delay for

nodes or maximum capacity for edges) can be specified to make the traffic model more complex, adapting parameters and interactions to more closely model a specific real-world system of choice.

3.8 Snapshot: a method for outputting the state of the entire simulation

Though the random factor prevents a truly reproducible simulation, simulation snapshot output has been designed in such a format that it can also serve as config file *input* for later simulation. This provides continuity between all files associated with the simulation and makes it easier to recover the simulation and its output in case of machine failure/crashing.

Snapshots output a human-readable dump of the entire simulation state, including all details on cars, edges, and nodes. It was deemed essential that the user be able to obtain these snapshots whenever they like, allowing the flexibility to save every single tick state (and maybe dump to a database for detailed analysis), or choose only final or important interim states if desired.

Chapter 4

Network Traffic Simulator: Structure and Architecture

This software is comprised of several interconnected modules. Each module represents an abstract concept required for modelling a traffic scenario, each containing as many classes as are needed to create the components necessarily for that concept. This results in the creation of three selfcontained modules representing the network, the cars/objects traversing the network, and the state-changer. To make the simulation complete and fully self-contained, the user may utilize two additional optional modules for generating the network structure and car objects. Following naming conventions, and module that a user may directly interact with has been named using capitalization; dependent modules (hidden to the user) are named using only lowercase.

4.1 Essential Modules for the Network Traffic Simulator

4.1.1 Network Module: traffic_network

As noted in the previous chapter, a network must have three components (network structure, nodes, and edges) *and* there exists an inherent hierarchy to these structures (a network only exists by defining sets of connected nodes). Though for creation purposes it may make sense to define the nodes and edges and let the network be a dependent object, that does not make sense for the problem at hand: traffic simulation is the analysis of objects moving over a network, therefore the Network itself must be given (requiring a class of its own), leading to the component Nodes and Edges being dependent

classes. The module **traffic_network** has been created to collect the instances and interactions of all network components for a simulation instance.

The resulting module structure is as follows:

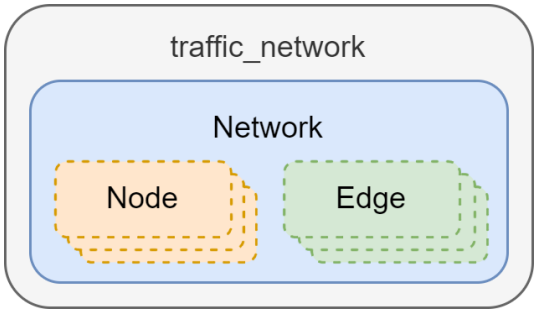

Figure 4.1 Hierarchical structure of objects within the traffic_network module

The **traffic_network** module creates and runs an instance of a Network object which creates and corrals its constituent Node and Edge objects. Please note that the functions in this module should be hidden from the user. Instead, the user should call for changes using the**TrafficManager** API.

Network class

The **Network** class contains all attributes and functions relating to the network as a whole. It contains a pointer to the TrafficManager simulation instance it was created for, a global timestamp, and dictionaries mapping IDs of the nodes, edges, and cars on the network back to the objects they represent.

Since the simulation depends heavily on the structure of the network, the **Network** class hosts a slew of functions whose output is utilized by both the User and cars running on the network. It has the capability to place new cars on the network, add and remove nodes and edges from the network, find (optimal) paths between any two points on the network, assess the consumed movement "potential" of the system, and oversees the movement tick() function on the **Node** level.

Node class

The **Node** class contains all attributes and functions pertaining to the purpose of a **Node** in the network. Each **Node** has dictionary mappings of its inbound and outboud **Edge** ids back to the **Edge** objects they represent. This is essential for facilitating the movement of cars across the **Node** when cars move off one **Edge** and onto another via the**Node**'s tick() function. **Node**s may also have an instersection_time_cost; this value is used to account for the real-world time (and space) delay that occurs when switching from one edge to another (like the physical time and distance of turning a corner on a busy road, or the time it takes to perform an http handshake).

Additionally, the **Node** supports stoplight capabilities, which are more broadly categoriezed as a time-based restriction on available **Edges**. Attributes like stoplight pattern, duration, and delay define which "open" and "closed" states are available, with a change_stoplight() function to control the cycling through these states.

Edge class

The **Edge** class contains information and functions related to an individual **Edge** in the **Network**. While it needs pointers back to the start and end **Node** objects that define it (and other attributes that were part of the **Edge**'s **Network** config), it also does a lot of work in the actual movement of cars in the simulation.

When an **Edge**'s start_node object calls on the **Edge** to tick(), the car must not only keep track of the **Car**s currently located on it (via a **Car** id to **Car** object mapping), but discern between which **Car**s it has already and has yet to move on the current tick (in order to prevent cars from moving more than their potential energy allows them to). For each **Car** on the **Edge**, the **Edge** must try to move it as far forward as possible, checking on the potential, the exit positions, *and* the physical location of any other **Car**s on the **Edge** to prevent the cars from overlapping or phasing through each other. Furthermore, the **Edge** collects a list of any cars that have left the **Network** (by path completion or User input) whos last position was on that **Edge**, ensuring that **Car** information is never lost.

The **Edge** also has another role to play in the **Node** tick() of its end_node:

when a **Node** tries to transfer a **Car** from one **Edge** to another, it may fail if there is no room available on the new **Edge**. This means that the old **Edge** must be capable of both holding onto the **Car** in case of failure, or handing it off if successful.

4.1.2 Car Module: network_cars

"Car" is the general term used in this document (and the Simulator itself) to describe an object traversing the network as it allows for intuitive labeling of its attributes. Once instantiated, a car is not dependent on the network to continue existing; to represent this semi-independence, the Car class was moved to a separate module:

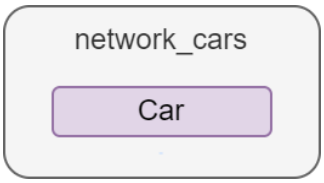

Figure 4.2 Classes structure within the network_cars module

In practice, though, the car is not very interesting when trying to simulate overall network behavior and ensuing traffic scenarios. Any attributes the user may care about (such as current location) are only relevant in context. So while **network_cars** technically exists as a self-contained module, it is never used in isolation. Instead, this module is automatically imported into the **traffic_network** module, seamlessly allowing these two modules to interact with one another.

This module creates and stores a car object. A car is created when called into existence by an API call via the **Traffic** module. While the **network_cars** module is fully dependent on the **network_traffic** module to move, car objects can exist separately. Thus, **network_cars** is imported into the **network_traffic** module to allow for object-network interaction.

Once again, the functions here should be hidden from the user. Instead, the user should call for changes and additions using the**TrafficManager** API. Internal functions belonging to the **network_traffic** module can be seen below:

Car class

The **Car** class contains all pertinent static information about a Car (like id or car_length), as well as information and ability to update dynamic values (like route_status). The flexibility of the **Car** to handle both types of data is what allows the Simulation to remain flexible and extensible.

Most notably, a car can be assigned a "Static" or "Dynamic" type; a "Dynamic" type indicated that the **Car** is capable of recalculating its path any time time it reaches a **Node**. This is coupled with a route_preference attribute which determines what kind of path ("Fastest", "Shortest", or "Random") the **Car** will follow when it enters the **Network** or recalculates when reaching a **Node**.

The **Car** must also keep track of its state at all times. Any time the **Car** moves during an **Edge** tick, it must recalculate its remaining potential to know if it is eligible for movement again. It must also be aware of where it is trying to exit the **Network** so it can tell the **Edge** where to exit, and must also know if its eligible to move at all (the User may halt it with a pause_car(Car_ID) function).

It is important to note that while the **Car** stores all of the information needed for it to run in the simulation, the simulation exists without the **Car**, and thus the **Car** itself has no power to change any **Network** attributes.

4.1.3 State Module: Traffic

The final component necessary to creating a simulation is a mechanism for advancing the state of the network. State changing includes adding, removing, and advancing any cars on the network as far as possible within a particular unit of time, and are all essential for creating a hands-off simulation.

But simulation state refers to more than the set of current car locations. It includes system metadata (like lists of nodes and edges in a network, and their attributes) and dependent calculations from that metadata. By allowing the user an access point to adapt any component of the network, this software achieves its goal of being adaptable and extensible to other types of networks and simulations.

The Traffic modules serves as an API to the underlying simulation, allowing users to (indirectly) interact with the network components and car

objects. The set of all these access points into the simulation allows for the direct management of traffic and has thus been wrapped into an aptly named class, **TrafficManager**:

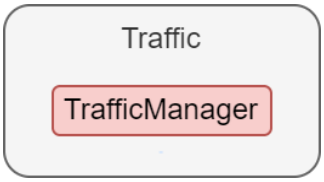

Figure 4.3 Classes structure within the Traffic module

Note that the **Traffic** module allows only for indirect access to the simulation components. By using this API as an intermediary between users and network simulation components, the user is given access only to commands that are relevant to analysis, and hide internal functions that facilitate those actions. For example, if a user wants a particular car to halt in place, they can call on the API function that requests it. The **Traffic** module then passes that request to the **traffic_network** and/or **network_cars** module to handle if and when it becomes relevant.

TrafficManager Class

The **TrafficManager** class instantiates a simulation instance and exposes all necessary/desired functions for interacting with the simulation once it is running. This means that it must keep track of the **Network** is controls and a global timestamp for the system. The timestamp (which can be retrieved with a get_timestamp() function) allows the User (or **CarGenerator** module) to know when to trigger an event like add_car(Car_ID) or output the simulation state with get_snapshot().

As the **TrafficManager** class serves as the simulation's API, it contains various functions for the User to interact with the simulation or passively return network information. Besides add_car, the User may use **TrafficManager** to remove_car, or even pause_car (and subsequently resume_car). The User may also request **Network** information via the **TrafficManager** class for things like listing all possible routes in the network, expected time (or distance) to complete a particular route, or a dump the simulation's state via get_snapshot (which the User may then want to save in a json file or otherwise).

4.2 Essential Module Interaction

Putting the modules together, we get the following depiction of how the modules interact with one another:

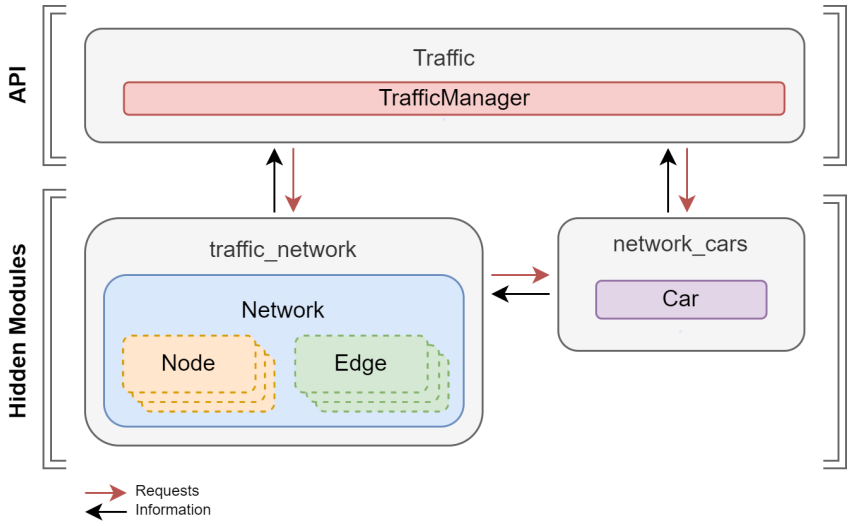

Figure 4.4 Full view of interactions between the modules and their individual components

However, as the user doesn't need to concern themselves with the specifics on which functions in which classes work and when, we can streamline the architecture diagram to:

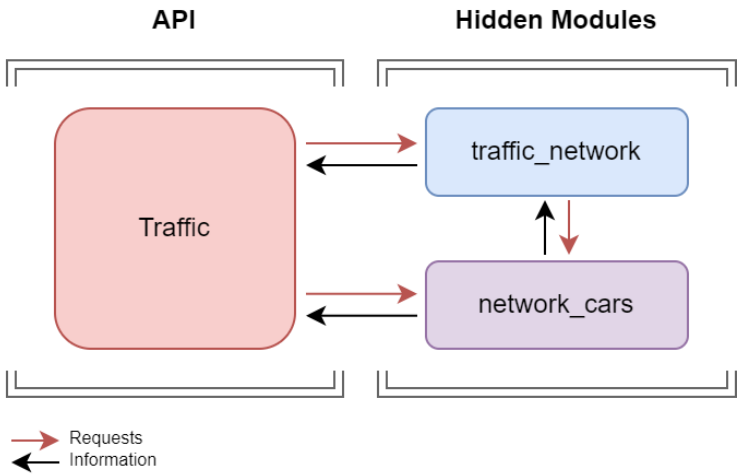

Figure 4.5 Generalized overview of interactions between modules in the TrafficManager ecosystem

4.3 Extended Module Interaction

For the simulation to run, it must be provided with car objects to move on the network and a network object to move the cars along. How this information is provided to the Traffic module is left up to the user, but some suggestions are provided below.

4.3.1 Importing Cars

With existing traffic data, one may want to create a realistic simulation by generating cars and adding them to the network in a way that emulates the real-world data. To do this, a colleague has created a separate **CarGenerator** module that allows users to generate cars probabilistically. This optional module can be run on its own, or integrated directly into the simulation by passing along the **TrafficManager** and **Network** instance pointers to the generator. The details of the generation process and types of patterns the module generates can be found in a colleague's project writeup.

In lieu of using the **CarGenerator** module, users may provide their own custom car objects (as a dictionary) as input into the **TrafficManager** instantiation. By allowing file-import flexibility to the car adding mechanism, the simulation is therefore capable of using its own snapshot outputs as input to a new simulation. This allows users to run the same batch of cars (created manually, or by the module) to be run over multiple simulations and compare outputs.

4.3.2 Importing a Network

Much like importing cars, flexibility has been given in how network structures can be loaded into the simulation.

An optional module, **UnderlyingNetworkGenerator**, has been written for creating networks based on mathematical concepts like Erdős-Renyi random networks or complete bidirectional networks. This module creates a stripped-down version of a **Network** object, assigning only the most essential attributes to individual nodes and edges. Simple underlying networks may be beneficial for simulations where emphasis is to be placed on the mechanisms of the network action itself (like stoplight cycles or road capacity metering) rather than microscopic analysis. Though the current version of this module is quite bare-bones, it can (and will) be easily adapted to include more complex,

probabilistic attribute assignments.

For real-world simulations, though, a user would probably prefer to import existing road or network data. This can be done by converting geojson/csv/shp files to the json format seen in the simulation repository's file "*EXAMPLE_network_config.json*". This method has been tested and confirmed by taking road and waterways WFS data from Het Nationaal Wegenbestand [MvIeW22] and stripping the "road" segments to just their ID, start-point identifier, end-point identifier, and directionality (if a segment was labeled as bi-directional, then the segment was duplicated with a new ID, reversing the start and end points).

4.3.3 Complete Module Interaction

Incorporating the car and network imports and user direction into the Simulation ecosystem, we end up with the resulting User-Software Architecture Model:

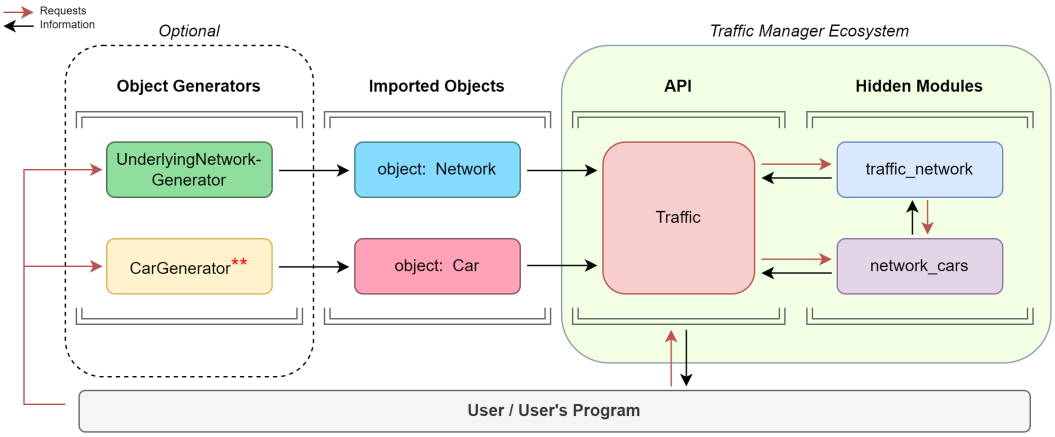

Figure 4.6 Generalized overview of interactions between modules, including the use of optional generator modules and/or load files. The starred component (created by a colleague) is separate from the traffic manager ecosystem detailed in this paper

Chapter 5

Using the Network Traffic Simulator

This chapter illustrates how the traffic simulator software works by guiding the reader through one hypothetical road-traffic simulation step by step in sections 5.1 through 5.4. Though automotive traffic is used for the example case, instances where the given car simulation may differ from other types of network simulations have been highlighted.

5.1 Define the scope of the simulation

The details of setting up the simulation depend on several factors:

1. What type of network system is being modeled?
2. What is the goal of the simulation?
3. What types of objects traverse the network? Are the uniform?
4. What data is already available to create the simulation? If none, what do we know about the system?
5. What changes or additions need to be made to a basic network structure to model the idiosyncrasies of the network or desired observations?
6. How long will the simulation run? Or how what criteria needs to be met before the simulation is complete?

For our walkthrough example, the answers might look like the following:

1. A province's road network. Some roads are one-way, and some roads have multiple lanes per direction.
2. We want to analyze if adding more highway lanes reduces traffic during rush hour.
3. Cars and shipping trucks share the road.
4. There is geospatial data available for city roads (in csv format). However, there is only one lineitem per named road segment. We don't have car data, but we know that during the daily rush hour period around 8000 cars travel northbound.
5. Trucks tend to go 10% slower than the speed limit. Everything else seems normal or standard enough.
6. We want to observe the simulation for a window of time 1 hour before to 1 hour after rush hour. We will need to run two simulations (one including the extra highway lanes and one without).

5.2 Identify or create the required data

This software requires that network and car objects be imported as a dictionary objects. At the bare minimum, each imported network requires 2 or more nodes and 1 or more edges (with start and end node IDs specified), with unique identifiers for each. Each imported car requires (at minimum) a unique identifier and a start and end location for the journey it will take. Users may specify additional attributes that align with their simulation scope, but these are not essential for a simulation to run.

Users feed information into the simulation by providing specially-formatted objects (json or otherwise). A complete view of the file format can be found in section 7.3 (Appendix), but the main thing to note is that very few fields are mandatory.

The Network config file consists of two parts: Node list and Edge list. The only mandatory field for Nodes is the *id* as a unique identifier, but users may also add details on stoplights/passage gates (if applicable). Edges have a few more required attributes (*id*, *start_node_id*, *end_node_id*), but also allow the user to specify conditions for metering like speed and maximum capacity. Real-world simulations will typically require the user to specify *edge_length*

as well, though the simulation will run regardless, defaulting to equal-length edges (if none specified).

5.2.1 Object creation

Users can create the objects themselves, or use the optional **CarGenerator** and **UnderlyingNetworkGenerator** modules to output such files. These config files are necessary to launch a new simulation, but are also the expected format of any additions to the network or simulation during the run time. Any objects imported after initialization should only include new objects.

Because the simulation has been designed to be capable of importing objects from another simulation's outputs, it should be noted that the config files can be combined into one document.

5.2.2 Example

Back to our simulation example. we identified an existing dataset for network that needed formatting and a schema (but no data) for creating cars. This requires reformatting the csv network data and using the **CarGenerator** module (described in a colleague's thesis) to create the cars.

To use the network csv data, we first need rename and reconfigure the fields to match those required to match the our scope, then convert the file from csv to json. In this case, the steps required are:

1. Rename the start and end node fields, and label the edge identifier field as "id".

2. Redefine speed. If the daytime speed limit is 60 km/h but we want our system to have a tick size (process new state) of 1 second to improve simulation accuracy. Converting to meters per second, that max_speed is now 16.667.

3. Create mirror edges for segments labeled "bidirectional". If edge 1789 is bidirectional and goes from A to B, then we need to create a new edge with a new unique ID that goes from B to A.

4. Duplicate edges to represent the multiple lanes. If the highway is currently 4 lanes wide, then 3 additional copies are needed for each edge representing the highway.

5. Create a second copy of the network file, but add extra edges representing the extra lane on the freeway.

6. Convert the csv(s) to json.

If we decided to create cars manually instead, we could use the DEFAULT car config file to to batch out the creation of two car types: cars (who travel like normal), and trucks (who drive 10% slower, which translates to setting a default max_tick_potential to 0.9).

5.3 Set up a new simulation environment and translate the scope to code

The **Traffic** module serves as an API into the simulation controls; to use **Traffic** and either of the optional generator modules in the Traffic Network Simulator, the user must import them into their workspace.

Set up a new python file with the following (or something similar), downloading/installing the modules and (default) config directory associated with this software.

In your working file, use relevant calls to the **Traffic** API to build your simulation. Generally, this includes building a script that calls for a certain amount of ticks, adding cars at relevant points along the way. For more complex simulations, the user may want to use commands to pause or resume the motion of particular cars (simulating traffic accidents) or prevent access to entire roads/sections of the network. Please refer to the chapter 7, *"Appendix"*, for a full list of API commands available in the current software version.

For our working rush hour traffic example, we have the following things to consider:

- Import the car and (two) network config files and store them as dictionary objects. Note: The car objects are assumed to be stored in file after generation here, but that is not necessary.

- Since we are running 2 simulations (the existing road network with and without the additional lanes), we need to instantiate one TrafficManager per simulation.

- Determine how many ticks the simulation(s) must run for. If "rush hour" is 2 hours and we want to also capture the hour before and hour after, we need our simulation to run for 4-hours' worth of ticks. Since the max_speed precision is defined as meters per second, the simulation should run for 14400 ticks.
- Batch out when cars enter the network wince the 8000 cars obviously do not all start driving at the same time. Perhaps for the first half hour, 400 cars enter, 800 the next half hour, 1500 for the next four half hours, then 400 for the last 2 half hours. (Though there are more accurate ways to model network enter time than these discrete buckets, that choice is left to the user).
- Store network snapshots. One snapshot per tick may be excessive, but a snapshot every minute may be reasonable. This could translate to grabbing a snapshot every 60 ticks.

To translate these steps to code, we can loop TrafficManager.tick() for as many ticks are needed between snapshot dumps (TrafficManager.get_snapshot()) and car additions (TrafficManager.add_car(Car_ID)). For an example Simulation working file, refer to section 7.4 (Appendix).

5.4 Interpret the simulation outputs

When running tm.tick(), a pair of statements will print in the terminal for each tick:

- "Steps needed to process tick: n"
- "Percent of available energy used on tick: m.00%"

Due to how the tick function works, the full set of nodes and edges will processed as many times as it takes for no more energy to exist or the tick to cause all cars to use up their energy potential. This number of iteration will be 1 if no cars are able to move at all (due to having completed their journey or being labeled temporarily by the user as "immobile"); but more frequently the minimum count is 2 due to a second pass checking if any more motion is possible. Any number higher than this is a proxy for how complicated the current state change is, but holds no direct meaning on its own.

However, the second item holds more weight. Since cars move by using tick potential, if something is preventing a car from moving, movement potential will remain at the end of the turn, indicating that some kind of bottleneck or obstruction is in the way. Due to the random order of node and tick processing, the user should refrain from taking the energy consumption as a direct metric of congestion, but instead use a moving window average to detect trends or even a global average when comparing simulations (if evaluating the difference a capacity meter, or speed limit change, etc can make).

Of course, the user can also elect to save snapshots of the whole network simulation at any point in time. These snapshots can be individually analyzed for a detailed overview of current network state, or aggregated with a database to observe or compare individual (or categories of) cars and edges.

Chapter 6

Next Steps

As this is the first version of the Network Traffic Simulator software, there are several aspects that can be added or improved upon in successive iterations. Some features are already hinted at in the code itself with placeholder functions, and others don't involve the existing code at all. This version succeeded in building a functional and extensible traffic simulator that, due do its modularity, can be easily tweaked and expanded by users or future contributors. Generally, the improvements fall into three categories: software improvements, feature expansion, and creating a visualizer.

6.1 Software Improvement

software design-wise, the system is solid in that its logic is predictable and airtight, and that the system is extensible. However, there are still many changes that can be made to improve the simulator, particularly in terms of computational complexity.

1. **Subclasses**: Though the use of subclasses is at odds with the design considerations outlined earlier in the paper, the current use of **if** statements to check for particular properties is not completely in line with the principle of extensibility. To remedy this, the use of subclasses should be considered for nodes (stoplights/none) and for cars (static versus dynamic path-following).

2. **Tick processing**: Currently, each tick cycle processes all cars on every single node and edge in the network, and cycles are repeated until no mar cars move. This means that the edges without cars are checked on each pass, and cars that are immobile or finishes moving for the

timestamp are checked with each network tick cycle. At the cost of increasing memory complexity to store extra variables and statuses, some drastic improvements in computational complexity can be made by doing so.

3. **Logs instead of prints**: While the two status messages printed per tick are beneficial for debugging and analyzing short simulations, they clutter the terminal and bury other print statements (like Exceptions) that may be more important. The next version will include an option for the user to toggle which kinds of print statements they want to see (car additions, car trip completitions, tick data, etc) and specify a log file to dump them to.

6.2 Feature Expansion

Building on the existing framework, the following features can and should be added to future releases:

1. **Remove Node/Edge**: While the simulator currently supports the addition of nodes and edges into a running simulation, there is only a placeholder for removal so far. This was due partly in the interest of time, but also due to the question of how cascading effects should be handled:

   - If an edge is removed, what happens to the cars that were on that edge or waiting to enter the network along that edge?
   - What about if a node is removed? Should this remove all inbound and outbound edges associated with the node?
   - Should cars lost in the removal be flagged with the generic "Removed from the simulation at X' status, or would something else provide more insight to the users?
   - Should removed edges be catalogued like removed cars are?

   Ideally, other users should be surveyed for input before any solution is proposed.

2. **Stoplights**: Currently stoplight change logic is available in the **traffic_network** module and it defines the active stoplight setting as the set of edges currently allowing cars to exit. However, this logic has not yet been activated during the node tick process. Before adding a check for

stoplight presence and status, additional input is requested from users whether the existing stoplight state (at the node level) makes sense at all.

3. **Smarter path calculations**: The current software version includes the option to select the best route based on shortest completion time in normal circumstances, but does not include any option for recalculating around traffic jams/congestion. This is done somewhat on purpose as a PhD student is working on a related project on metering, creating reinforcement learning models that identify these (literal) roadblocks and circumvent them. Additionally, before a "Fastest_now" calculation can be created, a more consistent and intuitive metric for congestion levels is needed.

4. **Congestion Metric**: Currently there exists a proxy for network congestion levels (it's the "Percent of available energy used on tick" print statement for each tick call to the **Traffic** module. While it would be simple to consider the used energy metric per individual edge, there is not enough evidence (yet) that this is a useful or insightful metric. We must consider the following:

    - Currently congestion is considered in terms of total energy consumption and on the network level. But would it make more sense to report on the car level (the number of individual cars that were able to move their full distance, the number that were stuck at a complete standstill, and/or the average tick potential used by cars that used some in-between amount)?
    - Or should congestion be considered as the average of edge-level congestion?
    - Are we even on the right track associating congestion with movement potential? Perhaps we should define it in terms of the ability of a new car to enter the network: What is the available remaining capacity per edge?

6.3 Visualization

Currently, the only way a user can view the output of a simulation is by saving and viewing snapshots. These snapshots were explicitly designed to be used for generating a simulation visualization, but the visualizer itself

was not yet created due to time constraints. Below is a proposal for building a visualizer:

- **Simulation precision**: As the simulation runs in discrete tick intervals, it makes sense to update the visualization with each tick, which can be done by utilizing the "get_snapshot" function already created for this purpose. While this does not create a continuous simulation, true continuity is not achievable due to the code design. However, continuity can be approximated by using animations to transition the image from one tick state to the next.
- **Snapshot deltas**: In the current version of the code, you can find a placeholder function labeled "get_snapshot_deltas", which would report only the *differences* between one snapshot and the next (rather than dump data for the entire network). Creating this function would allow a much smaller file to be sent to whatever server would host the visualizer, though is entirely non-essential for local instances or small simulations.
- **Implementation**: Because the visualizer will be built from the snapshot json files, the language that the visualizer is written in doesn't need to match the simulator. Instead, we can use JavaScript to take advantage of the **vis.js** library or **D3.js** library, each specifically written to display (dynamic) graphs.
  - vis.js provides simple graphs off the bat. While it doesn't seem like it (easily) supports the ability to add objects (cars) to the edges, you can change the displayed elements to reflect attributes weights. This would allow you to create a visualization displaying congestion, where the thinkness of the edge between two nodes is proportional to congestion. This can be beneficial for visually isolating traffic bottlenecks. D3.js appears very much the same.
  - Another option for simulations based on real-world geographic data is to utilize the ArcGIS API. While this does not solve the issue of displaying cars on the network, it does accurately reflect the relative relations and distances between and two nodes, whether they are connected or not.

Chapter 7

Appendix

This sections contains excerpts from the Network Traffic Simulator code and an overview of the system this software was created and tested on. The version you see here is the complete documentation for version 1.0.0, released 1 July 2022.

The features found in this version were created to meet (and exceed) the requirements and expectations set by Universiteit Utrecht staff and serves as a graduation thesis project for their MSc Applied Data Science program. This release, while a bit barebones on the generator side, sets up a complete, adaptable framework for urban traffic simulations (and other use cases!).

Please check the project's repository (below) for the latest instantiation:

`https://github.com/julialruiter/Traffic_Simulator`

7.1 Dependencies

So much as possible, libraries have been kept to the standard Python libraries. This means that the current Network Traffic Simulator software version has no external dependencies.

7.2 System Specifications

This software was created and tested on a machine with the following specifications. Though no official stress-testing has been done for this publication, this information may be relevant in analyzing any metrics reported in follow

ups:

OS Name	Microsoft Windows 10 Home
OS Version	10.0.19044 N/A Build 19044
OS Configuration	Standalone Workstation
OS Build Type	Multiprocessor Free
System Manufacturer	ASUSTeK COMPUTER INC.
System Model	VivoBook_ASUSLaptop X521EA_S533EA
System Type	x64-based PC
Processor	Intel64 Family 6 Model 140 Stepping 1 GenuineIntel 2803 Mhz
Installed RAM	16.0 GB (15.7 GB usable)

7.3 Example Config Files

This section contains the full expected config file structure referenced inchapter 5, *"Using the Network Traffic Simulator"*. Mandatory fields for the configs have been bolded:

7.3.1 Car object format

```
{
    "car_list": [
        {
            "id": <int>,
            "start_edge": <int>,
            "start_pos_meter": <float>,
            "end_edge": <int>,
            "end_pos_meter": <float>,
            "path": <list of consecutive edge ids>,
            "car_length": <float>,
            "car_type": <"Static" | "Dynamic">,
            "route_preference": <"Shortest" | "Fastest" |
                                "Random">,
            "max_tick_potential": <0 < float  1>
```

```
        },
        ...
    ]
}
```

7.3.2 Network object format

```
{
    "node_list": [
        {
            "id": <int>,
            "intersection_time_cost": <int>,
            "stoplight_pattern": <list of int lists>,
            "stoplight_duration": <int>,
            "stoplight_delay": <int>
        },
        ...
    ],
    "edge_list": [
        {
            "id": <int>,
            "start_node_id": <int>,
            "end_node_id": <int>,
            "edge_length": <float>,
            "max_speed": <float>,
            "max_capacity": <int | Infinity>
        },
        ...
    ]
}
```

7.4 Example Simulator Setup Code

To translate the Simulation example from chapter 5, *"Using the Network Traffic Simulator"* into code, we may write the following:

```
from Traffic import TrafficManager
import json
```

```
if __name__ == "__main__":

    # import configs
    network_config_original = None
    try:
        with open("./configs/road_data.json") as
                  original_json_file:
            network_config_original = json.load(
                                      original_json_file)
    except Exception as E:
        print(E)

    network_config_newlane = None
    try:
        with open("./configs/road_data_extralane.json") as
                  newlane_json_file:
            network_config_newlane = json.load(newlane_json_file)
    except Exception as E:
        print(E)

    car_config = None
    try:
        with open("./configs/generated_cars.json") as car_file:
            car_config = json.load(car_file)
    except Exception as E:
        print(E)

    # batch cars
    cars_batch_1 = car_config["car_list"][0:400]
    cars_batch_2 = car_config["car_list"][400:1200]
    # ...etc

    # set up the 2 simulations
    tm_original = TrafficManager(network_config_original)
    tm_newlane = TrafficManager(network_config_newlane)

    # advance timestamps for simulation 1, adding cars when
          necessary, outputting snapshots when necessary
    for car in cars_batch_1:
        tm_original.add_car(car)
```

```
for tick in range(1800):     # 30 min
    tm_original.tick()
    if tick % 60 == 0:
        with open(str(tm_original.get_timestamp()) +
                      '_snapshot.json', 'w') as f:
            json.dump(tm_original.get_snapshot(), f)

# ...etc for remaining car and time batches
#  repeat "advance timestamps" for tm_newlane
```

Bibliography

[D+22] German Aerospace Center (DLR) et al. SUMO user documentation, 2022. https://sumo.dlr.de/docs/index.html.

[Dag94] Carlos F Daganzo. The cell transmission model: A dynamic representation of highway traffic consistent with the hydrodynamic theory. *Trans. Res. Part B: Methodol.*, 28(4):269–287, August 1994.

[GPK02] David Goldsman, Sebastien Pernet, and Keebom Kang. Logistics 1: simulation of transportation logistics. In *Proceedings of the 34th Winter Simulation Conference: Exploring New Frontiers, San Diego, California, USA, December 8-11, 2002*, pages 901–904, January 2002.

[IH11] Teerawat Issariyakul and Ekram Hossain. *Introduction to network simulator NS2*. Springer, New York, NY, 2 edition, December 2011.

[LC08] Kun-chan Lan and Chien-Ming Chou. Realistic mobility models for vehicular ad hoc network (vanet) simulations. In *2008 8th International Conference on ITS Telecommunications*, pages 362–366, 2008.

[LWB+18] Pablo Alvarez Lopez, Evamarie Wiessner, Michael Behrisch, Laura Bieker-Walz, Jakob Erdmann, Yun-Pang Flotterod, Robert Hilbrich, Leonhard Lucken, Johannes Rummel, and Peter Wagner. Microscopic traffic simulation using SUMO. *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*, 2018.

[MH15] Ajith Muralidharan and Roberto Horowitz. Computationally efficient model predictive control of freeway networks. *Transportation Research Part C: Emerging Technologies*, 58:532–553, 2015. Special Issue: Advanced Road Traffic Control.

[MvIeW22] Rijkswaterstaat Ministerie van Infrastructuur en Waterstaat. Dataset: Nationaal wegen bestand (nwb), 2022. data retrieved from PDOK, https://www.pdok.nl/introductie/-/article/nationaal-wegen-bestand-nwb-.

[NBKL21] Mohamed Nahri, Azedine Boulmakoul, Lamia Karim, and Ahmed Lbath. A reactive system for pedestrian mobility simulation. *Procedia Computer Science*, 184:469–475, 2021. The 12th International Conference on Ambient Systems, Networks and Technologies (ANT) / The 4th International Conference on Emerging Data and Industry 4.0 (EDI40) / Affiliated Workshops.

[SKMR14] Guilherme Soares, Zafeiris Kokkinogenis, José Macedo, and Rosaldo Rossetti. Agent-based traffic simulation using SUMO and JADE: An integrated platform for artificial transportation systems. pages 44–61, November 2014.

[vdGPvA19] Jeroen P T van der Gun, Adam J Pel, and Bart van Arem. The link transmission model with variable fundamental diagrams and initial conditions. *Transportmetrica B: Transport Dynamics*, 7(1):834–864, December 2019.