Approximately Orchestrated Routing and Transportation Analyzer: City-scale traffic simulation and control schemes

Dustin Carlino

December 10, 2013

Abstract

Along with other intelligent traffic control schemes, autonomous vehicles present new opportunities for addressing traffic congestion. Traffic simulators enable researchers to explore these possibilities before such vehicles are widespread. This thesis describes a new open source[1], agent-based simulator: the Approximately Orchestrated Routing and Transportation Analyzer, or AORTA. AORTA is designed to provide reasonably realistic simulation of any city in the world with zero configuration, to run on cheap machines, and with an emphasis on easy use and simple code. Experiments described in this thesis can be set up on a new city in about five minutes. Two applications are built on AORTA by creating new intersection control policies and specifying new strategies for routing drivers. The first application, intersection auctions, allows humans to instruct their autonomous vehicle to bid on their behalf at intersections, granting them entry before other drivers who desire conflicting movements. The second, externality pricing, learns the travel time of a variety of different routes and defines a localized notion of cost imposed on other drivers by following the route. This information is used to tax drivers who choose to improve their own trip by inconveniencing others. These two systems demonstrate AORTA's utility for simulating control of the traffic of the near future.

[1] `http://www.aorta-traffic.org`

For the impatient –

Acknowledgments

First, I'd like to thank my advisors, Dr. Peter Stone and Dr. Stephen Boyles. They have dedicated their time and expertise in helping me explore traffic, agent-based simulation, and game theory. They have granted me the freedom to try what I wanted and the right amount of guidance to publish. I thank the Astronaut Scholarship Foundation for their generous financial support and for the opportunity to meet Captain Robert Crippen. I'd like to thank Dr. Michael Quinlan, Piyush Khandelwal, and others involved with the Autonomous Vehicles Freshman Research Initiative stream for providing an environment for starting AORTA. Thanks to Mike Depinet and Axel Setyanto for the occasional hack-a-thon during the early FRI days, and to Michael Levin for being the first real user of AORTA. Thanks to Todd Hester for doing a splendid job teaching and for introducing me to Clarke taxes, Vickrey auctions, and mechanism design. I'd also like to thank Dr. Calvin Lin for luring me away from traffic simulation for a semester. May your points-to sets remain sound.

Next, I'd like to thank my friends Colin Christ, Mandy Gabriel, and Jason Wolfe for consuming the devil's avocado, rubber-duck debugging, and putting up with my ramblings throughout college. I'd like to especially thank my friend Holly Hansel for designing the beautiful AORTA logo featured on the title page, and my roommates Sarah Labianca and Grace Hansen for keeping my sanity meter lubricated with their dry humor.

Finally, I'd like to thank my parents for their love and support throughout my life. I thank my dad especially for teaching me to drive, and my mom, who tried (and failed) to teach me a sense of patience.

Contents

List of Tables

List of Figures

1 Introduction

Anybody that drives in a city needs few statistics to motivate the search for solutions to traffic problems. In 2012, drivers just in the United States spent 5.5 billion hours, burned 2.9 billion gallons of fuel, wasted $121 billion in lost time and fuel, and emitted 56 billion pounds of carbon dioxide while stuck in traffic [18]. Building more roads to increase capacity takes money and space that many cities lack, so cities must instead utilize existing roads more effectively.

The advent of autonomous vehicles (AVs) has the potential to change these issues. Universal adaption of reliable AVs means decreased following distances on freeways, the virtual elimination of the $30,000$ deaths per year in the US [1] caused by drivers, a radical shift in lifestyle as monotonously-spent hours are returned to commuters, and the introduction of new legal and ethical dilemmas. These changes are just the beginning: current traffic control systems – namely, intersections controlled by traffic signals and routes picked from driver experience or GPS recommendation – are built for human agents. **Intelligent transportation systems** such as real-time travel information, variable speed limits, and ramp metering have been introduced. Widespread adoption of AVs further catalyzes new systems that could route vehicles away from congestion, time the arrival of a hybrid car at intersections to allow for hyper-miling, coordinate flow between multiple intersections, and have traffic respond pre-emptively to approaching emergency vehicles.

Car manufacturers are already competing to deploy AVs to the public. By 2020, Tesla, GM, Audi, Nissan, and BMW all aim to market vehicles that are nearly fully autonomous [22]. Widespread adoption is probably still at least a few decades away. To explore ideas for controlling traffic of the future now, **traffic simulations** are used. There are many existing open-source simulators already. In 2012, we introduced a new open-source simulator: the Approximately Orchestrated Routing and Transportation Analyzer, or AORTA [4]. This thesis heavily revises and expands on that paper. In addition, this thesis explores several novel ITS control schemes. Three high-level goals have directed AORTA's development and distinguish it from other simulators:

1. Reasonably realistic simulations on real city maps with fast setup
2. Real-time, user-friendly usage on cheap machines

3. Simple, high-level code that is easily extended

These features mean a reader with minimal knowledge can import any city into AORTA and start visualizing real-time simulations in about five minutes. Aside from replicating the experiments in this thesis, researchers can then quickly prototype new ideas by modifying AORTA. The code-base – a mere $11,000$ lines of code in a modern, multi-paradigm, Java-based language called Scala – has been completely overhauled several times throughout its history, and the end product is simple and succinct. Finally, two non-goals of this thesis should be stressed. First, some research in ITS deals with the hardware responsible for controlling the new behavior. Second, any real deployment of AVs and systems built on top of them requires the highest level of security and fault analysis. This thesis disregards both issues, presenting all ideas as prototypes.

The rest of this thesis is organized as follows. First, §2 evaluates other traffic simulators are evaluated. §3 then covers AORTA's map model. Next, §4 presents the simulator structure and traffic model, with §5 then focusing on three configurable pieces designed particularly for ITS experiments. With the simulator fully described, the fun begins. The first application built on AORTA is the notion of intersection auctions, described in §6. The second application, located in §7, influences drivers' routes with tolls designed to prevent congestion. Finally, §8 describes a number of future projects that could be built with AORTA, and §9 concludes.

2 Related Work

This section gives a brief overview of traffic simulators, focusing on a few that are most closely related to AORTA.

Wang and Prevedouros [21] provide an excellent overview of traffic simulators, classifying them by several factors. One characteristic is the time model. Some simulators run in **discrete time**, meaning agents repeatedly react to the world every fixed time-step. **Discrete event** simulations can pass time faster, as they jump to the time at which interesting events occur, such as an agent reaching the end of a road or a faster agent coming into range of a slower agent. These simulations develop formulas for determining when these events occur. Analytical models – not simulations – are often **continuous time**, meaning the traffic state at *any* moment can be determined by solving some differential equations. AORTA is discrete time because it allows for the greatest flexibility in agent behavior.

Level of detail is another major characteristic of a simulator. AORTA is **microscopic**, meaning individual drivers are modeled as agents. Microscopic simulators sacrifice speed for more detail, with each driver deciding a safe speed to travel and when to change lanes. Some microscopic models frame the environment and vehicle behavior in a cellular automata model [13], but these are quite dissimilar from the agent-based approach. In contrast, **macroscopic** models aggregate information at each road and intersection, typically as a number or density of vehicles. These simulators can be used to analyze much larger regions with millions of drivers. Finally, **meseoscopic** models hybridize the two, representing packets of vehicles at a time. These packets have a detailed behavior as in microscopic simulations, yet they represent many drivers at once.

This section only discusses open-source simulators. AORTA's purpose is to prototype new traffic control schemes, a task that requires modifying a simulator. As such, it is not useful to compare AORTA to simulators without the same degree of extensibility. Two popular closed-source traffic simulators must be mentioned, though. CORSIM is a detailed microsimulator, supporting multiple vehicle types and public transportation. It has a separate model for freeways and urban areas. PTV VISSIM is another widely-used simulator, with particular support for multi-modal transportation.

2.1 SUMO

SUMO [12], the Simulation of Urban Mobility, is one of the most widely used microscopic simulators. This section makes no attempt to explain all of its features; rather, its 0.18 release (the most recent as of October 2013) shall be evaluated against the goals of AORTA. SUMO has a high degree of configurability, mostly through XML files for demand data, routes, turn definitions at junctions, the map, traffic signal timings, and so on. More things are tunable than in AORTA without editing the source. However, its numerous capabilities come at the cost of a massive code-base of $125,000$ lines of C++. Upon rough inspection, the code is not easy to understand. For example, the function *isInsertionSuccess* in the *MSLane* class attempts to introduce a new driver into a lane, avoiding a crash. The function is 210 lines long with over 5 levels of nesting and conditional in some sections. A quick glance at the details reveal many basic blocks that should be refactored as separate functions.

SUMO also has a high startup cost. Using provided tutorials, I tried to run a simulation using an OpenStreetMap encoding of Seattle. The map importer lacks reasonable defaults: the option to remove disconnected portions of the map is off by default. A separate Python script is needed to generate random trips for agents. Even after disconnected sections of the map were pruned, simulating these generated trips resulted in immediate failure[2]. Undoubtedly there is a way to make everything work, but there is no simple set of instructions for getting started with a common use-case.

There has been no proper evaluation of AORTA against SUMO, but some conclusions can be drawn. SUMO is a much more powerful simulator, but it comes at the cost of a more complex code-base and a difficult user experience for beginners. AORTA replicates much of the core behavior in only $11,000$ lines of a higher-level language, and has scripts and reasonable defaults so that a beginner can begin simulating in any city within minutes. An argument may be made for the choice of C++ for speed, but there are doubts. Loading the map for Seattle in SUMO took longer than in AORTA, probably because the encoding in SUMO is uncompressed XML – a verbose $100MB$ – compared to AORTA's binary format of about $4MB$. Disk IO as a limiting factor appears to have been neglected in this design decision.

[2] An agent could not find a route for the trip generated by the Python script

2.2 MATSim

MATSim [2], the Multi-Agent Transport Simulation, is another major traffic simulator. Its focus is on the bigger picture of evaluating how people choose different aspects of their trips – where to go, when to leave, what route to take – rather than simulating detailed traffic conditions. It has two traffic flow models, the newer of which uses discrete-event time. Equations give times at which links will change state (meaning a driver exits or enters a road), and effects like congestion spillback have to be explicitly modeled. In AORTA, such effects arise naturally as each agent constantly reacts to its environment. MATSim is agent-based in the sense that each driver updates its trip plan after getting a score from simulation. Modeling new control policies for intersections, for instance, would not be straightforward in MATSim. As such, further comparison is difficult. For reference, MATSim 0.5 consists of $140,000$ lines of well-organized and documented Java. The tutorials make it quick to start using, but getting visualization components to run takes more work.

2.3 AIM

The Autonomous Intersection Management (AIM) project [7] introduces an intersection control scheme designed for autonomous vehicles. As drivers approach an intersection, they request their desired movement from an intersection agent. This intersection manager determines when the driver can safely make the turn and sends back a reply. The manager predicts potential collisions by dividing the intersection into a grid and reserving space-time tiles for each vehicle. The buffer around a vehicle can be adjusted to account for mechanical inaccuracies, in case the autonomous vehicle cannot exactly follow its intended path.

AIM is not a general traffic simulator; instead, it focuses on interactions between vehicles and infrastructure at one intersection. AIM has been extended to cover a small network of linked intersections [10]. AIM is a large source of inspiration for AORTA: originally AORTA was conceived to explore the idea of running AIM's intersection protocol at every intersection in an entire city. This goal was never quite realized, as AIM's protocol is too expensive to simulate on one machine at thousands of intersections at a time. In addition, intersections in AIM have an idealized geometry, but AORTA uses more realistic data, which cannot as easily be divided into grids. Note

that an intersection policy inspired by AIM exists in AORTA; see §5.1.3. As a point of comparison, the AIM 1.0.3 simulator consists of $40,000$ lines of Java.

3 Maps

To simulate traffic, some environment for vehicles is needed. For example, AIM hardcodes a simple network of roads and intersections, while MATSim can run over all of Europe. In contrast, AORTA is designed to work at city-scale, simulating a metropolitan area. This scale is large enough to study interesting questions about coordination between drivers and intersections, while not being so ambitious as to compromise real-time usage.

There are three goals of AORTA's maps. First, we wish to run simulations on real cities. Handcrafting or randomly generating maps is a useful testing technique, but we wish to avoid the bias from using mock cities. (Note that we do not simulate realistic traffic demands in this thesis, due to the lack of data – see §3.5.) Second, we desire generality. Although it may be convenient to develop experiments for a familiar city, there is no reason to make decisions limited to one city. Results obtained in one city should be repeatable in another, and if they differ greatly, properties about the networks should be discovered to explain this. Finally, zero configuration should be required for new maps. Time is better spent developing code, not hand-tuning maps. As a trade-off, development time has been spent finding workarounds to the imperfect map data available.

3.1 OpenStreetMap

Based on the three goals of AORTA maps, OpenStreetMap (OSM)[3] is the source for map data. OSM is a project that releases maps of the entire world under the Creative Commons license. Anybody can contribute map data, and quality control is done on a volunteer basis. OSM files can be downloaded from the main website, although this process only works for a limited map size. Other websites[4] provides dumps of many major cities, and OSM also has a planet-wide file. Tools like JOSM[5] can be used to further refine a large map into the desired region, specified as a bounding polygon.

The OSM map model consists of **ways** and **nodes**. Nodes are a single latitude/longitude pair, and ways are a sequence of nodes. The nodes in a way encode an approximation of the center of a road. Ways have many annotations, describing the road's classification and one-way restrictions. OSM

[3] `www.openstreetmap.org`

[4] `http://metro.teczno.com/`, `http://download.bbbike.org/osm/`, etc

[5] `http://josm.openstreetmap.de/`

files also encode other information like the presence of points of interest and bike paths, which AORTA ignores. OSM files are XML and easily parsed.

3.1.1 Limitations

There are several deficiencies with OSM data from AORTA's perspective. First, OSM ways do not specify how many lanes are on each street. (There is a tag for number of lanes, but it is rarely used in most cities.) AORTA guesses the number from the classification of a way, which is occasionally wrong. The impact of the wrong number of lanes can be rather severe. If a major road with several lanes is marked as minor for one segment, then traffic will bottleneck there. This problem only occurs in simulation; in reality, the number of lanes is different, so congestion does not necessarily form there. In principle, mis-marked data like this could be fixed and contributed back to OSM.

Another lack of detail is in intersections, which OSM does not even explicitly encode. Two ways sharing a node is treated as an intersection. OSM does not describe the turns available. One-way roads are marked, so some turns can be prohibited. Since lanes are not explicitly encoded, turn lanes are of course not specified. If a major road feeds into a minor one and the number of lanes lessens, what can drivers in the rightmost lane of the first road do? They may go straight and merge with people from other lanes also going straight, or only turn right. For simplicity, AORTA never infers multiple lanes for turning left or right.

Overall, the limitations of OSM have heuristic workarounds. Familiarity with the city during development helps detect anomalies. The level of detail in OSM is really inconsistent[6], which is expected from a volunteer-driven mapping project.

3.2 AORTA Map Model

This section describes AORTA's map model and its limits, invariants, and encoding. An AORTA **graph** (or **map**) consists of **lanes**, **vertices**, **turns**,

[6]We once discovered that every tree on the east half of the Texas state capitol grounds is precisely marked. In our amusement, we reached out to the tagger of the trees to learn that her software had buggily erased the west side. Since then, somebody added in the west trees. It should be noted that entrance lanes to I-35 were missing in some places. Priorities are strange.

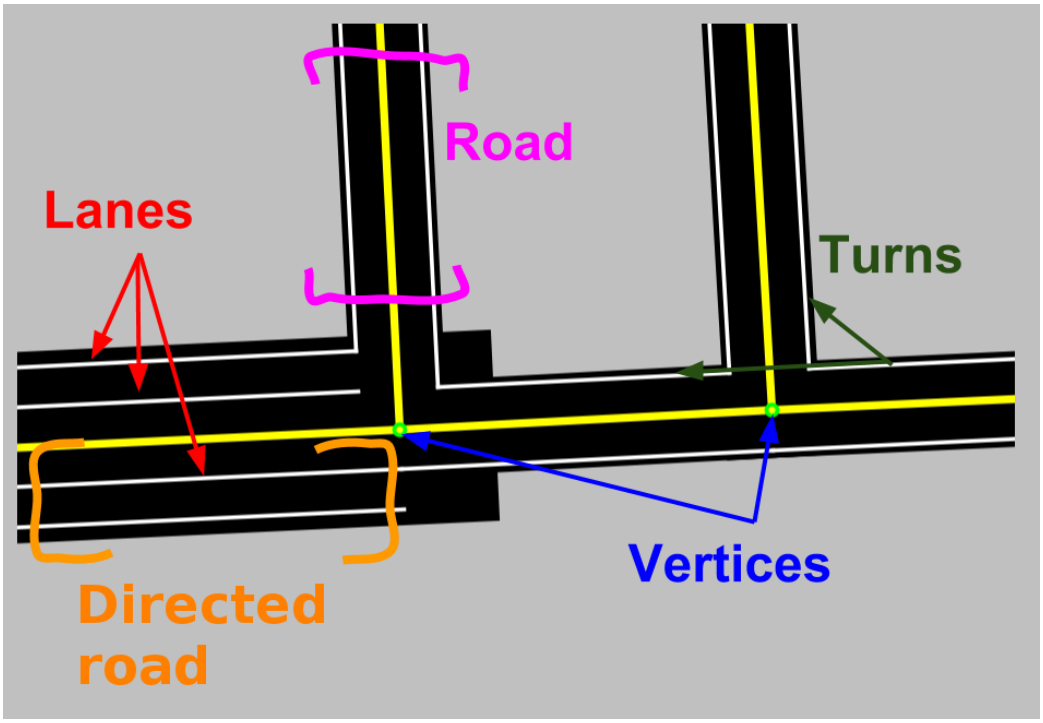

Figure 1: AORTA map model

and **roads**. Each road has two groups of lanes, one in each direction. One-way roads only have one group of lanes. The term **directed road** refers to each group of lanes. Vertices represent intersections, where roads cross. They contain turns, which connect a lane on a road incoming to the vertex to a lane on a road outgoing from the vertex. These elements are shown in Figure 1.

Drivers travel along lanes and turns in single-file. Turns and lanes are the same with respect to modeling movement. Geometrically, lanes consist of a sequence of line segments (when the underlying piece of the OSM way defined multiple points), and turns are a single line segment between the last point of the incoming lane and the first point of the outgoing lane. (It would be perfectly reasonable to model turns with curves, but this would only improve the visual representation and not any real simulation detail.)

For the purposes of finding paths, there are several ways to view an AORTA map as a graph. One interpretation is that from a lane, a driver may travel to any other lane in the same directed road by lane-changing or

to any turn connected to that lane. It turns out that it is inconvenient for a routing component to prescribe the specific lanes that must be used. Instead, directed roads are used as graph edges, with intersections linking them.

3.2.1 Map Properties

Some properties are always true of AORTA maps, guaranteed by the conversion process in §3.3. First, the directed roads of the map are **connected**, meaning that there exists a path connecting any two directed roads. If this was not true, it would be impossible for some drivers to find a route to their destination. In simulation, a driver may have to change lanes to follow a route. As described in §4.1.3, there are constraints on lane-changing involving the length of the road – namely, the driver must completely finish changing lanes by the time they are at the end of the road. There could be a case where a driver must cross 5 lanes to make a turn, but the road is not long enough to support this movement. To remedy this, the number of lanes a road has is limited based on its length, speed limit, and the lanechanging distance parameter.

A second property constrains the length of lanes. Because drivers stop 5 meters from the end of a lane in AORTA's model, lanes should be at least this long. In addition, the presence of short lanes implies multiple intersections spaced close together. In order to move through a section of the map, a driver must negotiate with each intersection. In practice, unless the intersections are coordinating in some way (such as synchronized timing on traffic signals), then unrealistic bottlenecks will occur near short lanes. To deal with these two problems, a minimum lane length of 50 meters is required. Merging short lanes into adjacent lanes enforces this limit, as described in §3.3.3.

3.2.2 Map Encoding

An AORTA map is characterized by the following data:

1 roads
- 1.1 ID
- 1.2 length (computable from coordinates, but cached for speed)
- 1.3 name
- 1.4 OSM metadata (type and OSM ID for debugging)
- 1.5 2 IDs of vertices linked
- 1.6 sequence of coordinates

2 lanes
- 2.1 ID
- 2.2 road ID
- 2.3 direction (called "positive" and "negative" arbitrarily)
- 2.4 lane number (an offset from the rightmost lane)
- 2.5 sequence of line segments
- 2.6 length (cached)

3 vertices
- 3.1 ID
- 3.2 location coordinate
- 3.3 turns
  - 3.3.1 ID
  - 3.3.2 origin and destination lane IDs

4 geometric metadata (width, height, x offset, y offset, scale)

5 map name

This data is encoded in a compact binary format. It could just as easily be expressed as XML.

3.3 Map Construction

This section details the process of converting an OSM map to an AORTA map. This process only has to be run once per map, not once per simulation.

3.3.1 Pass 1: Scrape data from OSM

Using a SAX XML parser[7], each XML element from the source OSM file is processed. Nodes correspond directly to a (longitude, latitude) coordinate. When a node is referenced by more than one way, it is added as a vertex. The road name, road type, coordinates, and other miscellaneous metadata are scraped from the OSM ways. Finally, coordinate normalization occurs. AORTA's coordinate system follows that of the Swing GUI library for convenience in drawing – the origin is at the top-left corner, only the positive quadrant is used, and coordinates are scaled to be in the range $[0, 5000]$.

[7]SAX parsers stream XML events. DOM parsers give a tree structure and are easier to use, but consume too much memory.

This max height/width is arbitrarily chosen. The min and max x and y coordinates are found, then all coordinates are translated and re-scaled[8].

3.3.2 Pass 2.1: Split ways into roads

OSM ways correspond to roads that cross many intersections. To move towards having a graph structure, this pass splits ways into roads that go between exactly two intersections. The list of points in each way is traversed, and each pair of control points contributes a road. Control points are either points marked as intersections in the previous pass (because more than one way references them) or the first or last point of the way.

3.3.3 Pass 2.2: Merge short roads

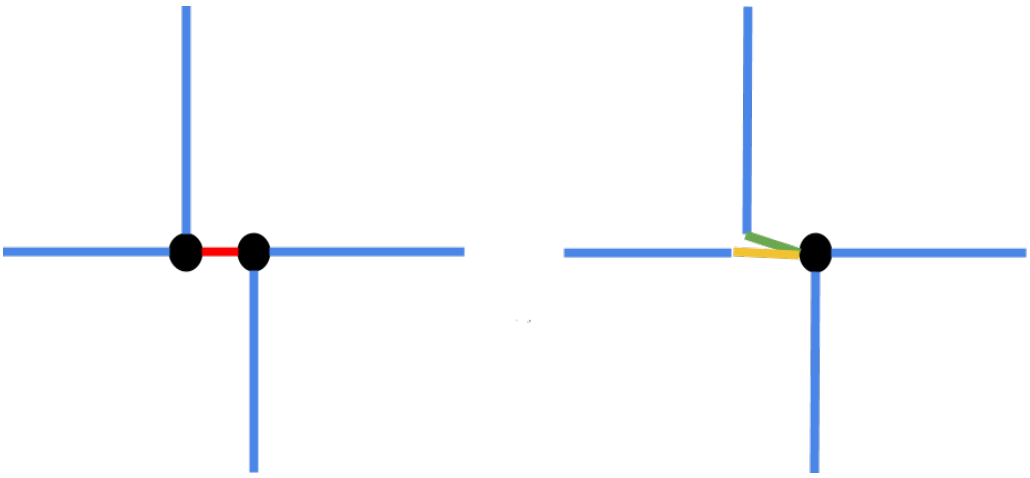

Figure 2: A short road (left) and its removal (right)

As mentioned in the map properties, short roads cause problems and must be removed. It is simplest to do this in pass 2, while roads are still directionless and have no lanes. This pass iterates over all short roads under 50 meters excluding cul-de-sacs[9], removes the road, and arbitrarily removes one of the road's two vertices. It then stitches up every road referencing the removed vertex, making it point instead to the other vertex. Figure 2 shows

[8]It would be simpler to leave coordinates in GPS-space and only transform in the GUI. This will be fixed in the future.

[9]roads that form a self-loop on a vertex

this process. On the left, the red road is too short. On the right, one vertex has been removed and both connected roads extended.

3.3.4 Pass 2.3: Collapse degenerate vertices

Degenerate vertices have only two roads leading to them. They are degenerate because no turns could ever conflict. The only purpose of keeping around a degenerate vertex is to keep data between the two roads separate – speed limits, number of lanes, or the name of the road. This pass finds all degenerate vertices, then merges the two roads and removes the vertex. This pass must be run after merging short roads in pass 2.2. Since 2.2 also removes vertices, it makes some vertices that were originally normal become degenerate. Thus, this pass should run after 2.2, to clean up any new degenerate cases, as well as pre-existing cases.

3.3.5 Pass 3.1: Multiply roads into lanes

Each road has a number of lanes in each direction. Directions are arbitrarily labeled either **positive** or **negative**, and a one-way road only has positive lanes. The number of lanes is given by a very rough heuristic based on the OSM road type[10].

3.3.6 Pass 3.2: Filling out turns

At this point, lanes link to and from vertices, but no turns connect them. This pass creates turns for each vertex. If there is only one road leading to a vertex, it is a dead-end. Assuming the road is not one-way, corresponding lanes are linked together with turns. Figure 3 demonstrates these U-turns.

At each vertex, every pair of incoming roads and outgoing roads is analyzed. The angle between arbitrary representative lanes on each road is calculated. If the angle is less than a crossing threshold of 18° or if the two roads came from the same original OSM way, then the connection will be straight crossing turns. If the number of incoming lanes equals the number of outgoing, then each corresponding lane will be linked by a turn. If there are less lanes on the other side of the intersection, then the rightmost incoming lanes will all have to merge into the one rightmost outgoing lane. If less

[10]Residential streets, service roads, and links have 1 lane; all other types have 2

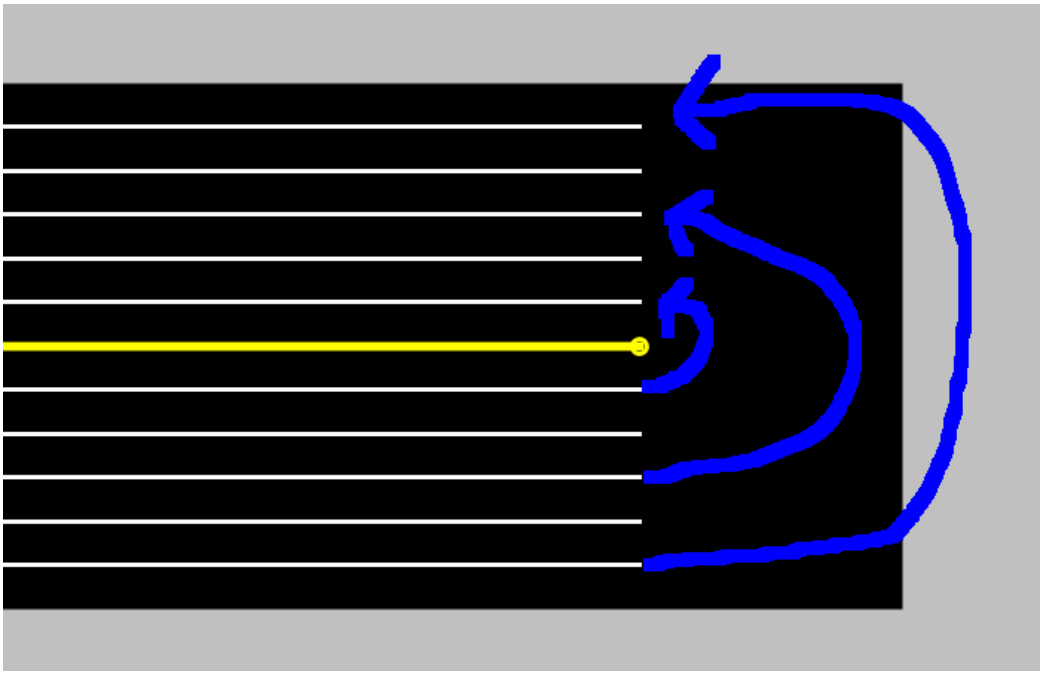

Figure 3: U-turns at a dead-end

lanes lead to more, then the leftmost incoming lane gets to pick from the extra choices of outgoing lanes.

When the angle between the two roads is beyond the threshold, then either the leftmost lanes are linked, or the rightmost lanes are linked. Multiple left or right turn lanes are never created.

3.3.7 Pass 3.3: Disconnected graph pruning

Lanes have **predecessors**, other lanes that lead to them via turns, and **successors**, other lanes that are reachable from them via turns. At this point, some lanes may lack one or both, since the prior pass may fail to fill out some turns that might exist in reality. Since maps must be connected, this pass employs a fixed-point algorithm to guarantee connectedness. The algorithm cycles between two stages. In the first, it discovers lanes with no predecessors or successors and removes them. In the second, Tarjan's algorithm [19] is run on the graph of lanes and turns in order to find all strongly-connected components. All but the largest component are removed. Since each stage removes pieces of the map and potentially creates more

pieces of the map to be removed, the algorithm iterates until there are no changes. After all of this work, all lanes, turns, roads, and vertices are renamed so that their IDs are contiguous.

3.3.8 Pass 3.4: Trimming lane geometry

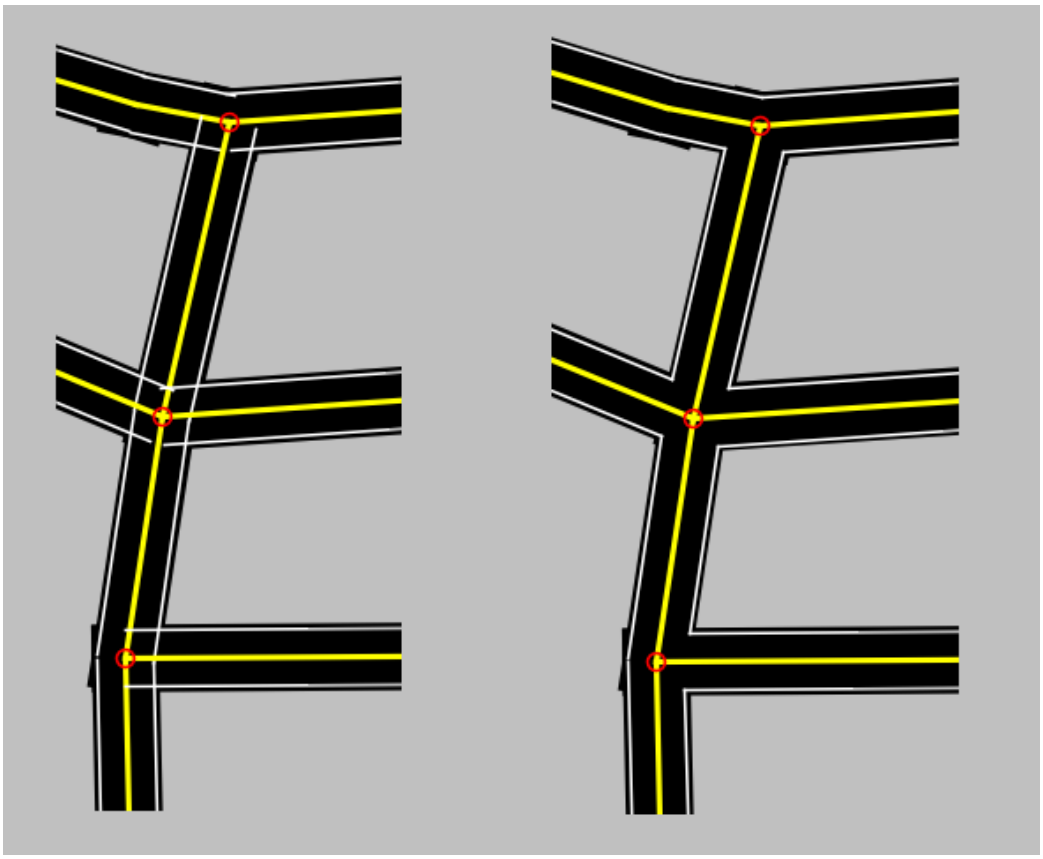

Figure 4: Lanes before trimming (left) and after trimming (right)

Due to the way lane lines are projected from road lines, lane geometry naturally extends until it juts into the intersection. This is visually displeasing, so this pass attempts to correct for this. The line segment closest to the intersection for every lane is quadratically compared. When segments intersect, both are trimmed back to the point of crossing. There may be many collisions for a given line, so the correction with the greatest magnitude is chosen. Figure 4 shows the results. Unfortunately, this fix makes

some correct lanes become too short, violating the efforts of pass 2.2. Since these transformations are aesthetic and 2.2's are semantic, for now this pass is disabled.

3.3.9 End of conversion

The built map is serialized in a binary format.

3.4 Evaluation of Map System

City	Austin	Baton Rouge	NYC	SF	Seattle
OSM file size (MB)	11	15	15	55	29
OSM nodes	40,677	48,608	50,461	251,581	118,237
OSM ways	3,214	6,421	2,446	7,833	5,337
AORTA file size (MB)	2.5	5.4	.9	5.3	4
AORTA roads	6,756	13,892	3,460	12,769	10,277
AORTA lanes	13,396	28,232	4,562	23,929	21,237
AORTA vertices	3,971	9,233	1,730	6,937	5,848
Time (mm::ss)	0:44	1:25	1:05	1:45	1:14

As the table above indicates, the one-time conversion process is cheap[11]. With a low-end machine (2GB RAM, 2.32 GHz chip), these map sizes lead to acceptable performance for most simulations. AORTA has not been evaluated on maps much larger than one city.

3.5 Demand Data

Once realistic maps are available, the next step is to produce realistic patterns of traffic, such as a morning rush from home to work. This requires **origin-destination** (OD) pairs indicating how many drivers leave from one region during some time to travel to another. There is a large body of work [14] describing various ways to produce these pairs. OpenStreetMap gives AORTA generality in maps, but unfortunately, there is no known source for demand data for many cities. (Recently, OpenStreetMap has annotated individual residential and business buildings in Seattle. Future AORTA work will harvest this data to generate drivers leaving from houses and going to shops.)

[11]Most of the expensive passes have low-priority simple optimizations to be done, so this process can be made even faster.

As a result, in all of the experiments described in this thesis, unless otherwise stated, the traffic demand is uniformly distributed. This means that if $10,000$ drivers are spawned per hour, their spawning time, origin, and destination are uniformly distributed from all available choices. This pattern is extremely unrealistic.

It may be possible to generate more realistic demands automatically, retaining generality. The process would involve picking a random road from the map, flooding outwards to encompass the surrounding region up to some threshold, and then defining roads in that region as popular origins and destinations. This process might backfire if the region happens to be an area with low population and few good routes to the region in reality, but it could offer some improvement over uniform distributions.

4 Traffic Simulator

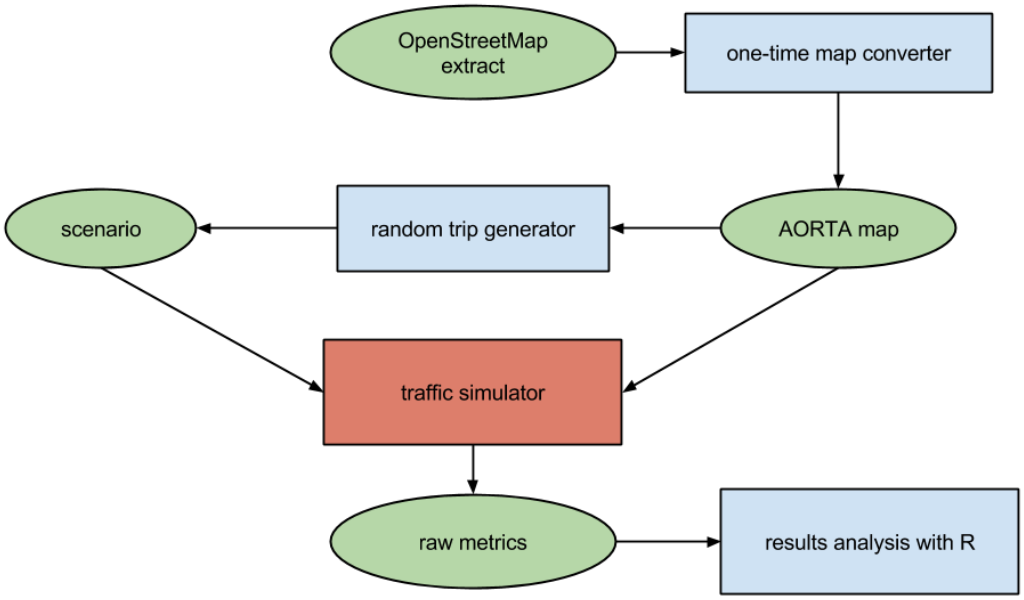

Figure 5: The interaction of the traffic simulator with other components

This section describes the main component of AORTA: the traffic simulator. Figure 5 shows how the simulator fits in the bigger picture of using AORTA. The remainder of this section dives into the simulator, starting with the traffic model in §4.1. Next, §4.2 discusses a major feature of the simulator: determinism. §4.3 then discusses gridlock, followed by an overview of implementation details in §4.4.

4.1 Traffic Model

AORTA is a microscopic, agent-based traffic simulator, meaning each vehicle is a separate entity that perceives its environment and responds accordingly. Time advances in discrete steps of 1.0 second. Space is continuous, meaning the front of a vehicle is located at a real-valued distance from the start of the road. (The alternative discrete space would split each road into tiles, with a vehicle occupying only one tile at a time.) Note AORTA includes no model for pedestrians, bikes, or public transportation.

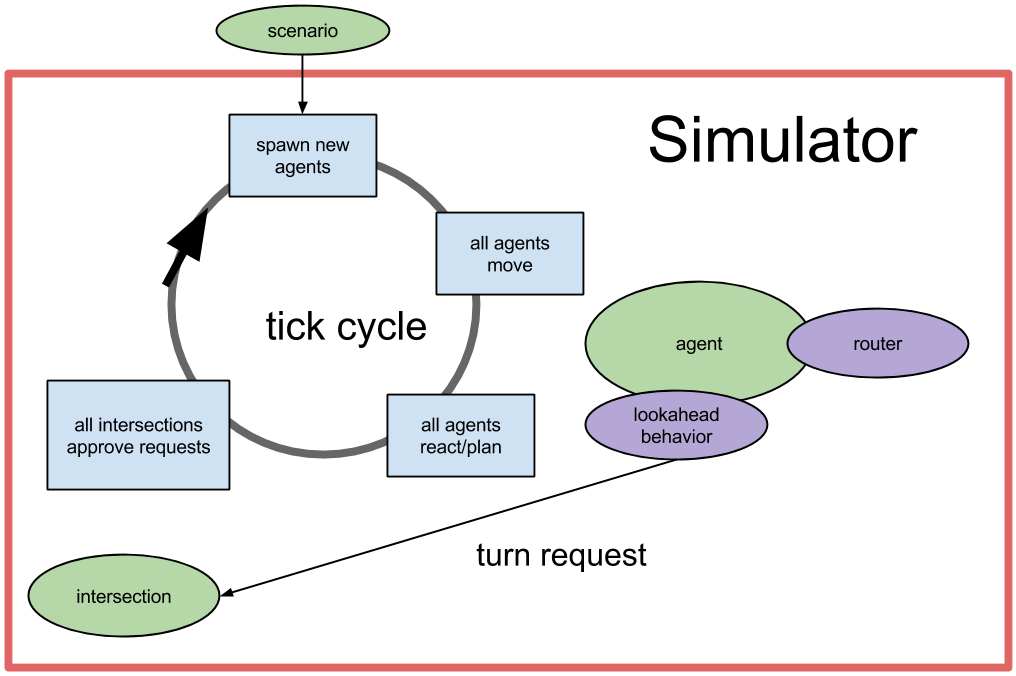

Figure 6: The main loop of the AORTA simulator

The simulation proceeds by repeating the steps shown in Figure 6. Specifically, every time-step – or **tick** – the following occurs:

1. New agents scheduled to spawn at the current tick join the ready set
2. All agents in the ready set attempt to enter the map
3. Action phase: Every active agent moves according to the target acceleration set in the previous tick

   Lane-changing is initiated before the agent moves during this tick
4. Optionally, all queues of agents verify no collisions have occurred[12]
5. Optionally, all intersections verify agents in the intersection are not performing conflicting movements, which would indicate a collision

[12] Agents present must retain the same relative ordering as the last tick. Due to lane-changing, agents can appear or disappear from the middle of the queue.

6. Planning phase: Each agent reacts to its environment, choosing a target acceleration to perform during the next tick

   Agents talk to upcoming intersections to request turns during this step

7. Agents finished with their route exit the simulation

8. Intersections react to the environment by linearizing the order of turn requests and granting leases to agents

4.1.1 Agent Lifecycle

This section gives an overview of each stage of an agent's life. Before a simulation starts, agents are scheduled to spawn somewhere at a certain time (see §4.2.2). To model leaving from driveways, agents start at 0 speed. To avoid existing agents colliding with them, several requirements must be met for some *new* agent to spawn on a certain *queue*:

- *new* cannot spawn in front of an agent a already on *queue* unless a could completely stop and avoid colliding with *new* in the worst case (which is *new* not moving at all)
- *new* cannot spawn too close behind another agent a on *queue*
- *new* cannot spawn in front of an agent a who has been approved by the next intersection to turn, since that could block a from turning and contribute to gridlock (see §4.3)

Once an agent is part of the simulation, they react every tick by setting a target acceleration for the next tick and optionally requesting to change lanes. Agents request turns from intersections before they may leave a road, as detailed in §5.3.1. Multiple agents may perform the *same* turn simultaneously, as long as they leave enough space between themselves. If the path of two turns could intersect at any point, then they are considered **conflicting** and cannot be performed concurrently. In Figure 7, the red turns conflict with the green turn.

An agent's destination is a directed road. When an agent comes to a halt at the end of any lane of the goal road, they vanish from the simulation.

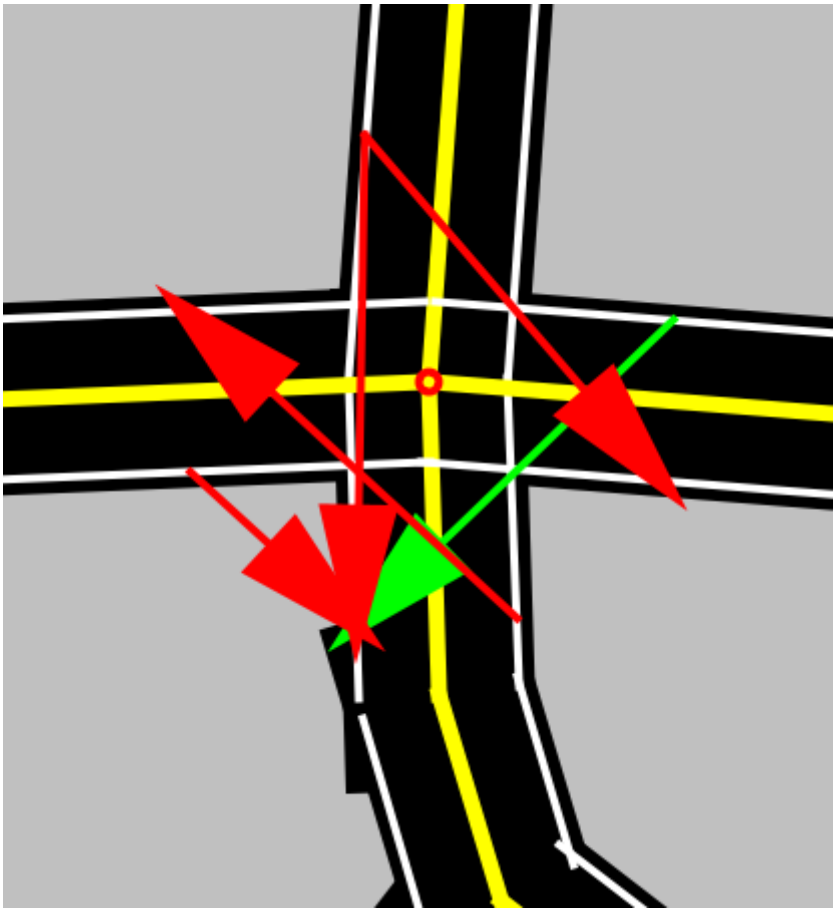

Figure 7: Turn conflicts

4.1.2 Parameters

The AORTA traffic model has a number of parameters that remain fixed during the course of a simulation. If, for instance, a run is saved and then later resumed, the same parameters must still be set for it to be meaningful. These parameters have not yet been tuned to match realistic driving.

Parameter	Description	Default value
dt_s	The duration of one tick	1 second
end_threshold	The distance from the end of a lane that an agent can stop to be considered at the end[13]	5 meters
follow_dist	The minimum distance that must be between every agent on the same queue. This represents vehicle length, since vehicles are otherwise treated as a single point at the front of the vehicle.	5 meters
max_accel	The maximum magnitude of acceleration/deceleration a vehicle can perform	$2.7\frac{m}{s^2}$
lanechange_dist_rate	The distance needed to perform a lane change is this value multiplied by the lane width of 0.5 meters	5

The tick duration *dt_s* determines how often agents react. Lower values result in slower-running simulations. When set high (at about 30 seconds), vehicles tend to lag at intersections. Drivers must request their turn during one tick, and they will not get a reply until the next tick. This delay maintains determinism of intersections approving drivers (see §4.2 for details). Consequently, drivers may spend 30 seconds on a short road just waiting for an intersection.

Note that a fixed follow_dist and max_accel imply that every vehicle is the same. It is a future task to introduce variety in vehicle capabilities and size. Some formulas in §5.3.1 rely on every vehicle knowing the max_accel of every other vehicle it encounters. When vehicles start to have variety, these formulas can remain simple by always using the worst-case value for these parameters.

4.1.3 Kinematics and Lane-Changing Model

AORTA uses a discrete model of time and models agent location as a distance from the start of a lane or turn. In each tick, agents move based on choices

[13]This threshold is set to produce a visual effect in the GUI of stopping before entering the intersection. It will be removed in the future to simplify the model.

made during the previous tick, then choose their desired lane-changing motion and acceleration for the next tick. This section describes how those choices are applied.

If v_i is an agent's current speed[14]) and a is the target acceleration, then $v_f = \max(0, v_i + (a * dt\_sec))$. There is a choice here: AORTA could assert that agents never reach negative speeds, or it could cap off high decelerations to let the agent come to rest, then not utilize the remaining duration of the tick. As the formula shows, the latter was chosen, for two reasons. First, floating point errors sometimes differ in the computation of what acceleration to use to stop and the computation of what final speed is set. Second, some formulas are vastly simplified with this choice in model. With this in mind, the distance traveled during one tick by an agent is given by $d = (v_i * t) + (\frac{1}{2} * a * t^2)$ where $t = dt\_s$ when $a \geq 0$ and $t = \min(dt\_s, -\frac{v_i}{a})$ when $a < 0$. The second argument to min is the time it takes to come to rest from an initial speed of v_i, from the formula $v_f = v_i + a * t$, with $v_f = 0$. This distance d first moves the agent along the current lane or turn. If the agent reaches the end, they move to the next turn or lane and and continue applying the remaining distance.

Many roads have more than one lane in each direction. Agents may need to change lanes to take certain turns; this is **mandatory** lane-changing. Alternatively, agents may wish to to avoid a lane with more queued traffic than others by performing **discretionary** lane-changing. Discretionary lane-changing simply prefers the lane with the lowest percent of its capacity filled. The process to lane-change is as follows:

1. During the planning phase of one tick, mark an adjacent lane as the target lane

2. During the action phase of subsequent ticks, if the agent is still on the same road, determine if it is safe to perform the lane-change

   If it is, set the lane-changing distance remaining to $lanechange\_dist\_rate * lane\_width$

   While lane-changing, the agent exists at the same distance along both the old and new queues. Agents in both queue have to account for an obstacle, and the agent changing lanes has to obey constraints imposed by others in both lanes.

[14]vehicles cannot move in reverse, so *speed* and not *velocity* is the appropriate term

3. Each tick while the agent is changing lanes, apply the distance traveled to the lane-changing distance remaining

4. When the agent has traveled the required lane-changing distance, end the motion, leaving the agent in only the new lane

Note that the actual lane-changing action is performed during the action phase, not the reaction phase. This choice was made so that all agents perceive the same view of the world during the reaction phase, so it could in theory be parallelized. The action phase, on the other hand, is sensitive to the order that agents execute their choices, since lane-changing safety requirements depend on the order of other nearby agents' lane-changing motions. This sensitivity is why it is vital to execute these actions in a deterministic order, as detailed in §4.2.

The safety requirements to start changing lanes are:

- The agent must finish lane-changing before reaching the intersection.
- Since the agent could not have requested a turn from the target lane yet (since agents cannot request a turn until they are sure they could follow through with it), they may have to stop by the end of the target queue to wait for their turn request to be approved.
- There must be a slot available on the target queue, for gridlock avoidance (see §4.3)
- The agent cannot move in front of another agent on the target queue that has already been approved to make their turn (see §4.3)
- If nobody would be behind the agent on the target queue, ask the intersection if anybody is scheduled to enter the target queue. If so, do not attempt to lane-change, since there could be a collision[15].
- Do not merge in too closely behind somebody else, or cause an agent on the target queue to crash. Check for agents in a "danger range."

[15]This check could be less conservative by checking the speed of the oncoming vehicle

4.2 Determinism

AORTA simulations are **deterministic**: given the same input, the same sequence of events will occur. This allows controlled experiments to be performed, so the effects of one change can be evaluated without worrying about other factors. Determinism also makes reproducing bugs easy. This section characterizes the input and give implementation details for how deterministic resimulation works.

Simulations often introduce noise to model realistic sensors and actuators that cannot perfectly perceive and act in the world. Noise is compatible with determinism by capturing and repeating the seed of the pseudo-random number generator used to produce the noise. In AORTA, noise could be introduced with human behaviors (see §5.3.2).

4.2.1 Implementation Details

There are several key details to achieve deterministic simulations in AORTA. First, a systematic use of data structures with deterministic iteration orders is important. This requirement is easily satisfied by using tree sets and maps, rather than hash-based structures. Second, all code relying on pseudo-random numbers must use the same seed in every run. There are not any uses of random variables in the traffic model itself; instead, some routing strategies (see §5.2) employ them. Breaking ties by lane IDs or agent IDs can introduce unintended biases – consider if intersections broke ties this way when choosing which turns to approve. Since agents that spawn earlier have lower IDs, this ordering could favor agents that have been traveling longer. If this effect is desirable, agent lifetime is explicitly available as a variable; relying on an implementation detail like ID is dangerous.

More generally, agents must perform their moves in a deterministic order. In a simpler traffic model, it could be possible for agents to plan their moves in any order (since they all observe the same fixed, atomic state of the world) and even for them to execute their moves in any order (since where they wind up is a function of the state of the world before anybody moves and their choice, with others' choices the same tick being irrelevant). Two details of AORTA's model prevent any order from being used. First, moves must be planned deterministically, because any action causing an agent to move into a different queue (whether by lane-changing or beginning/ending a turn) cannot be allowed without first allocating a spot in the queue for the agent,

due to gridlock avoidance (see §4.3). Second, lane-changing occurs during the action phase, not the planning phase. During the planning phase, agents indicate they wish to lane-change, and during the action phase, the first agent that can change lanes will get to do it. It is ongoing work to remedy these two cases, since being able to reorder agent moves is an easy way to parallelize these steps.

4.2.2 Scenarios

A **scenario** is the unit of determinism; it captures the full setup for a simulation. Simulating the same scenario twice produces exactly the same results. Specifically, a scenario encodes:

- the map on which a simulation is to be run
- every agent that will exist in the simulation

  their ID and the time at which they will spawn

  exactly where they start (the directed road and distance along it)

  their destination and the implementation-specific parameters for their route
- the configuration of every intersection

Scenarios are convenient structures for running experiments. For instance, measuring the effects of changing a congested intersection from a stop sign to a traffic signal is a matter of running two simulations. The second run would re-use the same scenario from the first, just assigning a different control policy to the intersection in question.

Scenarios can be created and modified using convenient command-line tools. Distributions of agents and intersections can be defined, so scenarios with some percentage of agents routing using a certain strategy can be set up. Using the GUI (see §4.5), regions of the map can be selected, and then agents can be spawned to travel from one region to another. Scenarios can be modified surgically, changing one agent or intersection, or at large, changing the behavior of all intersections.

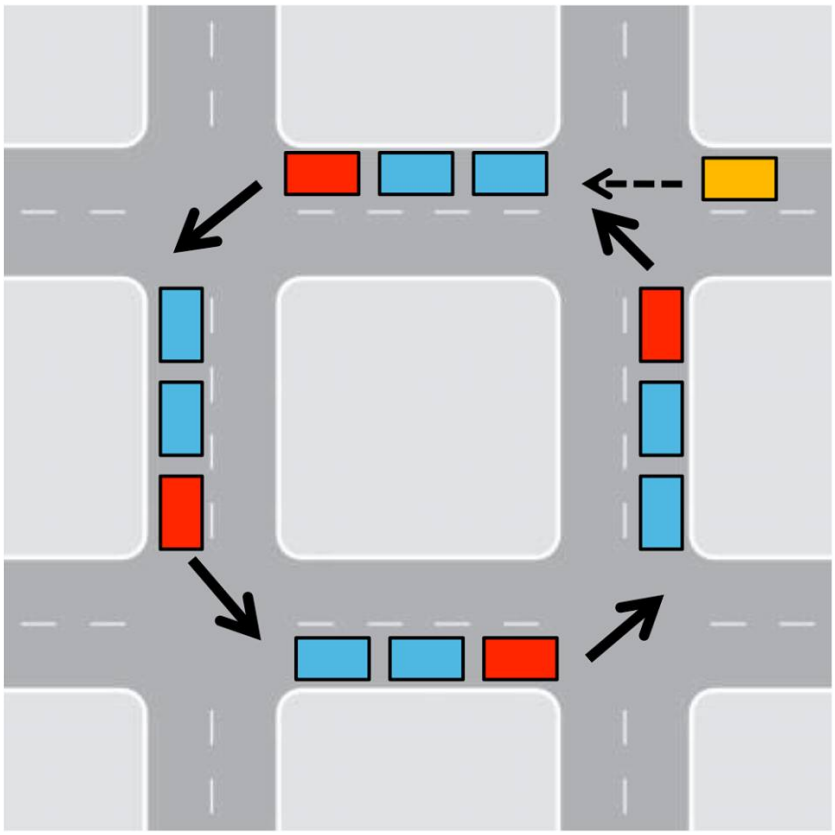

Figure 8: Gridlock at 4 adjacent intersections

4.3 Gridlock

A reasonable liveness property requires all agents to eventually reach their destination. This section deals with this property and how it is jeopardized by **gridlock**. Gridlock occurs when queues of drivers become interdependent. In gridlock, there are a sequence of full queues, with the driver at the front of each queue waiting to enter the next queue in the sequence. Until one of the drivers decides to take a different turn and move into a queue that is not full, all agents involved will remain stuck forever. Empirically, when gridlock in one section of the map occurs, other drivers trying to pass through that region will also become stuck, and the problem grows. Figure 8 demonstrates a localized form of gridlock. Each of the red cars wants to turn left. The yellow car wants to pass through the critical area, and by blocking forever, will cause the problem to grow. Queue spillback and gridlock are explicitly

modeled by some macroscopic traffic models, but it is an emergent effect in agent-based AORTA.

AORTA's intersections prevent agents from starting a turn if another agent is still performing a conflicting turn. So if an agent ever stops during a turn, then all agents wishing to perform conflicting turns become stuck. The solution is thus to prevent agents from starting a turn they cannot finish. In reality, humans would creep around one another, performing conflicting turns slowly, and gradually proceed. Nonetheless, gridlock is a real issue, and the techniques used to alleviate it in AORTA may be useful in reality too.

AORTA maintains some invariants to guarantee that once an agent's turn request is approved by the intersection policy, then nothing may prevent the agent from completing that turn. A **rest capacity** is defined for each lane, describing the max number of agents that can fit in the lane when at rest. (Reaching this capacity is not desirable, as no agent in the lane could be going the speed limit.) An agent reserves some of this capacity while they drive on a lane and just before they enter a lane, and they cannot be evicted by other agents. An agent allocates a slot on a queue in several circumstances:

- just before they spawn as a new agent
- just before they begin lane-changing
- when an intersection policy approves them for a turn, the turn's destination is marked
- when the reservation policy (see §5.1.3) chooses a turn request to interrupt the current set of approved turns, the interrupting agent allocates a slot

An agent frees a slot after exiting that lane. Note that the third rule prevents cases where agents on several incoming roads all desire a spot in one almost-full lane. The intersection will only approve agents up to the capacity of that lane. Also recall that one of the rules for lane-changing given in §4.1.1 prevents agents from lane-changing in front of another agent who has been approved for a turn. If this was not enforced, then an approved agent could be stuck behind an unapproved agent, violating the invariant.

This invariant does not fully prevent gridlock. Consider a case where all agents in some interdependent queues are stopped on lanes, with nobody in

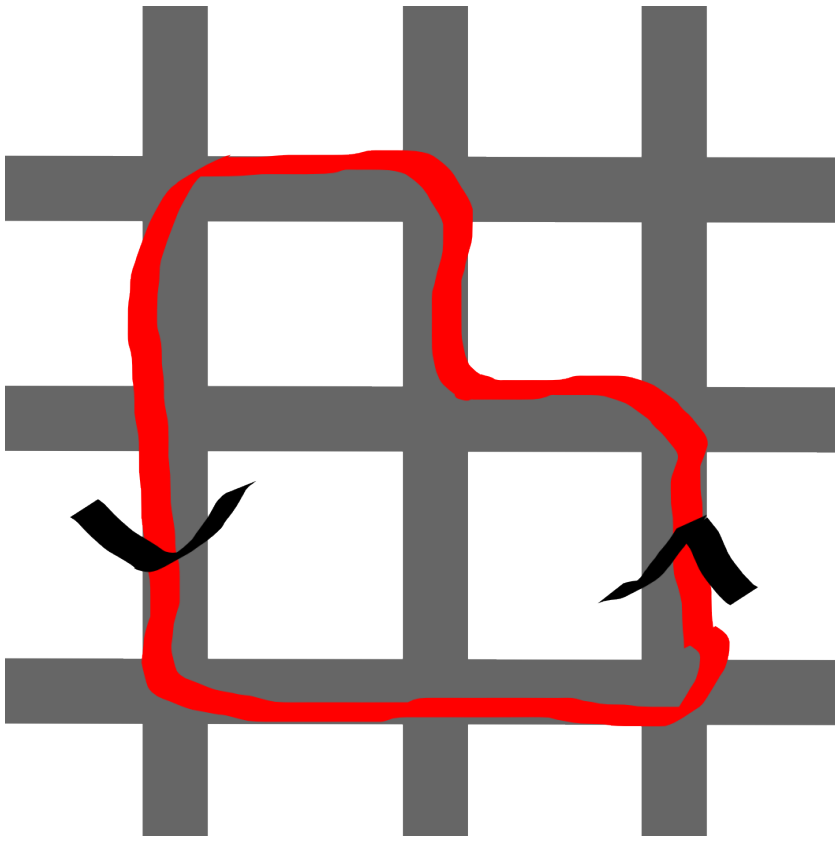

Figure 9: Gridlock at many intersections

any intersection. If the desired lane of every leading agent is already full, then gridlock is still present, as in Figure 9. One possible solution is preventative detection: before an agent requests a turn or before an intersection approves one, a cycle detector can discover interdependencies among the leading agent in full queues, so a different turn can be chosen. However, this fix is a bit too weak: there are some race conditions that can occur after approving but before starting a turn.

Instead, AORTA adapts a simpler approach: if an agent is waiting for its turn to be approved, the agent is part of a gridlock cycle, and the agent has another choice of turn, then the agent can reroute using congestion routing, described in §5.2.2. This way, the queues belonging to a gridlock cycle will gradually start moving again. This improves conditions, but does not solve every possible case.

4.4 Implementation

One of AORTA's selling points is its simple and concise implementation. This can only be fully appreciated by working with the code, but this section highlights some key features. The language choice is discussed in §4.4.1, then AORTA's organization is described in §4.4.2, and some specific design patterns utilized are listed in §4.4.3. In addition, note that AORTA is currently single-threaded, but parallelization is planned (see §8.2.2).

4.4.1 Scala

AORTA was originally written in Java, but it was completely ported to Scala after a semester of development. This section briefly argues why Scala was chosen. Scala mixes ideas from functional programming and object-oriented programming, allowing either style to be used. Since it runs on the Java Virtual Machine (JVM), Java libraries can be used from Scala.

AORTA particularly utilizes several of Scala's features. First, Scala distinguishes mutable variables from immutable values. Immutable data can be shared between parallel executing contexts without locks and enable some compiler optimizations. In AORTA, they are useful to organize the member fields of each class, accentuating the truly mutable state. Second, Scala lists have the functional programming transformation *map* and filtering $filter$ functions defined. Combined with anonymous functions, this replaces many verbose loops with a succinct one-liner. Third, Scala has a built-in option monad, which describe the return type of operations that may fail. Finally, Scala has powerful pattern matching that can branch to cases based on multiple factors.

While the default Scala compiler runs very slowly (about three minutes to compile AORTA's code-base), third-party build managers make compilation quite responsive. Specifically, *sbt*[16] incrementally compiles code. With sbt, the entire code-base can be compiled in about one minute, and most changes to the code only take a few seconds to recompile.

4.4.2 Components

AORTA is split roughly into three layered components: the map system, the microscopic simulator, and the GUI. The map system is independent

[16] `http://www.scala-sbt.org/`

of the other code, meaning that building a macroscopic simulator – or any other application – on top of the maps would not involve detangling much simulation-specific code. The microscopic simulator, though, is tied rather heavily to the map system. AORTA is not tied to OpenStreetMap though; maps from other sources could be converted to AORTA's format, as in §3.3. The third component, the GUI, is heavily tied to both the map and simulator. Although there is some generic code[17], most of the GUI visualizes specific details in the simulator.

Although AORTA has not been integrated with third-party software, its modularity should make this task easy, as long as the map model in the other package is compatible with AORTA's. As an example, the built-in assignment of traffic signal timings, described in §5.1.2, is simplistic and could be replaced with results from an external optimizer.

The following table gives the break-down of the 11,000 lines of code[18] in AORTA.

Package	Lines of code	Description
analysis	666	measuring stats, recording replays, and various experiments
common	549	configuration, physics formulas, serialization, and other general code
map	1000	map model
map/analysis	448	map pathfinding
map/make	1247	convert OSM to AORTA maps
sim	3362	traffic model
sim/market	525	intersection auction code
sim/policies	521	different intersection policies
tests	229	test cases
ui	1726	GUI

4.4.3 Design Patterns

This section lists a few particular patterns AORTA uses. First, each class is organized into several labeled sections for consistency. The mutable state of each object is grouped first, followed by standard methods to serialize and initialize the object. Then "action" methods that have a side-effect on

[17]Such as the *ScrollingCanvas* class, which exposes a large buffer and interprets keyboard and mouse controls to perform panning and zooming

[18]measured directly with Unix *wc -l*, which includes whitespace and comments

the object or other objects are defined. Finally, "query" methods that are read-only follow. It is useful to have documented whether or not a method is pure[19].

AORTA has many ID spaces to identify agents, roads, vertices, turns, lanes, and directed roads. Each ID is just an integer. It is easy to mistake one type of ID for another – there was once a bug in serializing routes where lane and directed road IDs were mixed up. When that bug was present, ID spaces were distinguished by variable naming or by context. However, since Scala 2.10 introduced **value classes**, this problem could be fixed with the type system. Simple classes like AgentID and RoadID are defined that just wrap integers. These classes form real types, so the compiler statically checks usages of these. Since types are explicitly declared in the code, naming the variable carefully is no longer needed to document the ID space. This sort of strategy is possible in any language allowing new types to be defined, but the reason to use value classes is efficiency. The boxed ID type is only known at compile-time; at runtime, the JVM just sees plain integers, so the JVM does not allocate new objects on the heap. This pattern can also be used to define physical quantities like distance, time, and speed.

AORTA makes heavy use of the **listener pattern** to maintain code separation. User interface or data logging code must respond when key events occur in the simulation code. Rather than littering unrelated code throughout the core module, events are broadcast. The event publisher does not care who consumes the events, enabling the decoupling. The other modules simply subscribe and consume the events.

4.4.4 Savestates

AORTA can take a snapshot of the traffic model at a certain time, encoding the exact internal state of every relevant object. This feature saves time. It may take several wall-hours to pass a few simulation-hours. Suppose some experiment introduces a change after several simulation-hours. Rather than simulate the baseline and modified scenario, one of the two can be simulated once, with a savestate captured right before the change is introduced.

All important simulation state is captured by the following fields. Some minor fields are omitted.

1 scenario name

[19]Scala currently lacks a standard way to annotate pure methods and statically verify they remain this way

2 tick

3 all agents

- 3.1 ID
- 3.2 route
  - 3.2.1 route type
  - 3.2.2 goal road ID
  - 3.2.3 rng
  - 3.2.4 for PathRoute only: the path stored, each chosen turn
- 3.3 RNG seed
- 3.4 wallet
  - 3.4.1 wallet type
  - 3.4.2 budget
  - 3.4.3 priority
- 3.5 start lane
- 3.6 position (lane/turn ID, distance)
- 3.7 speed
- 3.8 target acceleration
- 3.9 target lane ID (or -1 if none)
- 3.10 old lane ID (or -1 if none)
- 3.11 lanechange distance left
- 3.12 idle since (for calculating how long an agent has been waiting)
- 3.13 all tickets
- 3.14 turn ID
- 3.15 stats on when the turn was requested/accepted/finished
- 3.16 ID of each agent ready to spawn[20]

Agents' routes and wallets have multiple implementations, each with their own specialized state. The abstract Route or Wallet serializes an enum to mark the concrete implementation, then saves state common to all implementations. The subclasses each add their own specific state after that.

This tree of data is explicitly defined because built-in Java serialization is inappropriate for savestating. Some pieces of the object graph, such as the in-memory map model, are immutable and recreatable from the map; they do not need to be serialized with each save-state. Although such fields can be skipped during serialization with the *transient* annotation, these fields have to be restored when unserializing the object. Since manual intervention

[20]The details of the agent to spawn are in the scenario

is required anyway and reflection-based serialization is measurably slower, the tree of data is explicitly saved and used to recreate the traffic model. It takes about a half-second to write a save-state for most simulations using this method.

4.5 User Interface

While simulations can be run in a "headless" mode for maximum processing speed, AORTA also has a GUI for easy interaction. The map can be explored by panning and zooming, and there is also a way to teleport to a certain lane, road, or vertex. The GUI can highlight different routes, pause a running simulation, and track an agent as it moves. The user can draw a polygon around some region of the map, then capture all roads or lanes contained within to use as a zone for starting or ending trips.

The GUI uses the Java Swing library for rendering and GUI widgets. Rather than pollute the main code-base with GUI-related code, a rendering wrapper is created per object. The *DrawDriver*, *DrawRoad* (with a *DrawOneWayRoad* subclass), and *DrawLane* wrappers render each object, determine if it hits a bounding box representing the current view pane, and respond to being moused over. There is also a *ColorScheme* interface for determining the color of agents. The schemes work in optional chains: if a certain scheme does not prescribe a color to a certain agent, it delegates to the next scheme. The schemes, in order, include:

- *FocusVertexScheme* describes agents when the user mouses over a certain intersection. Agents not involved with that intersection turn gray, agents with an approved request are green, and agents with a request not yet approved are red.
- *CameraScheme* highlights in white the agent whose movements are being tracked.
- *StalledScheme* colors agents red if they have been at rest waiting to take a turn for more than 30 seconds. This pinpoints sections of the map with bottlenecks.
- *PersonalScheme* assigns any remaining agents to a randomly chosen color that they retain for the duration of the simulation.

5 Configurable Simulation Modules

To accommodate the exploration of autonomous traffic control, AORTA has three components designed specifically to be extended: intersection control policies, routing for agents, and agent behaviors. Note that AORTA does not have explicit support for vehicle-to-vehicle (V2V) communication. However, adding it to agent behaviors and modeling delayed or corrupted messages would not be hard.

5.1 Intersection Policies

An **intersection policy** controls how agents interact with a single intersection. Agents send a **ticket**, representing a requested turn, before they reach an intersection, and they may not begin their turn until the intersection approves their request. As described in §4.1.1, if two turns conflict, then no two agents may perform them concurrently, even if physically, there could be room for the interleaving.

Every tick, a policy examines all enqueued tickets and grant approval to some subset of them. As described in §4.3, policies must maintain invariants to avoid gridlock. If a ticket's agent is currently blocked from doing their turn, the policy must not accept the request yet. Agents may cancel a ticket and request a different turn, but this is discouraged. Currently, canceling tickets is only allowed for agents to avoid gridlock by navigating away from a stalled queue.

Each policy must choose how to prioritize requests. Any **intersection ordering** can be used with any policy. The default ordering is first-come, first-serve (FCFS). The first agent to request a turn is the first to be served. Since agents request turns far before they reach the intersection, each policy must form a list of valid **candidates** to serve. Often these candidates must be ready to start their turn in the next tick before they are approved. The other ordering, based on auctions, is the subject of §6.

One issue is how to assign policies to each intersection by default. The assignment is based on OpenStreetMap's annotation of the roads involved with the intersection as a minor or major road. In AORTA, intersections of minor roads have stop signs, traffic signals are placed between major roads, and a reservation manager handles crossings between a mix of minor and major roads. Other sources such as the Manual on Uniform Traffic Control Devices could be used to assign policies more appropriately.

The remainder of this section will describe each policy.

5.1.1 Stop Signs

The stop sign is the simplest intersection policy. It allows only one agent to perform their turn at a time[21]. Every agent must completely stop before proceeding. Each tick, if the intersection is not already occupied, the policy approves the first candidate ticket, according to the intersection ordering.

5.1.2 Traffic Signals

Traffic signals are the most common form of control for major intersections. Signals cycle through various **phases**, in which some turns through the intersection are permitted. Intersection ordering is used here to pick the next phase, not individual drivers. In AORTA's simplified model, each phase lasts a fixed *signal_duration* of 60 seconds. A signal cannot safely switch to the next phase until all agents approved in the previous phase finish their turn. As a result, signals need a heuristic to determine if an agent could finish their turn by the time the phase ends. Right now the heuristic is overly simple: if the agent travels at the road's speed limit and could start the turn by the time the phase ends, then the ticket is approved. There is much room for improvement here.

There are many choices for the phases. A massive body of literature on this subject covers everything from patterns of three-phase signals to multi-intersection light optimization. One simple but effective example draws inspiration from backpressure routing in communication networks [23]. In AORTA, there is currently only one simple greedy algorithm. A set of remaining turns to assign to a phase is initialized with all turns. To make a phase, one turn from the remaining set is chosen arbitrarily and removed. Turns from the remaining set are added to this phase in any order as long as no two turns conflict. This process is repeated, adding more phases as long as there are turns remaining. Finally, a post-process pass adds in all turns compatible with a phase, since only remaining turns were considered in the first pass.

[21] This is not realistic, as it prevents two non-conflicting turns from being done simultaneously

5.1.3 Reservation Managers

The reservation manager policy is designed exclusively for autonomous vehicles. It is modeled loosely after the AIM protocol [7]. It works by accepting one ticket at a time until it cannot accept anybody. The order is determined by intersection ordering, the candidates consist only of drivers who are not blocked by others whose tickets are unapproved, and each ticket is accepted as soon as no conflicting turn is scheduled. The first agent that requests a turn conflicting with the current batch is considered the **interruption**, and no new agents are admitted until this agent can start their move.

This policy's performance can easily be worse than that of traffic signals, if the ordering is poorly chosen. A major portion of this thesis is concerned with choosing the ordering using auctions; see §6 for more.

5.2 Routers

Each agent has a **router** responsible for leading the agent to its goal. As an agent moves between lanes and turns, it forwards its location to its router. In response, the router prescribes a sequence of directed roads to reach the goal. Sometimes, an agent cannot follow this route; it is an **unrealizable** path. This happens when moving from the current directed road to the next step requires changing lanes, but the target lane is full and cannot be entered in time. When this happens, the router must reroute. As described in §4.3, when an agent detects gridlock, it can also request that its router try to find a new path.

Aside from telling the agent which turn to request at an intersection, the router also prescribes a lane to use. This is necessary for informing the agent of a lane that leads to the next step along the path. It is also used for discretionary lane-changing by choosing the lane with the least congestion. One future addition could be avoiding unrealizable routes by having the driver change lanes several steps before a crucial intersection.

The remainder of this section will describe each router.

5.2.1 Dijkstra's Router

This router runs Dijkstra's algorithm [6] once, obtaining a table of distances from every directed road to the target directed road. Picking a turn amounts to hill-climbing: every turn leads to some directed road, so this router chooses

the turn leading to the lowest-distance next step. This router prefers lanes leading to the next step that are closest to the agent's current lane, since less lane-changing is easier. This router is simple, oblivious to traffic, has a moderate one-time time cost, and a reasonably hefty space cost ($O(|R|)$, with R being the map's directed roads).

5.2.2 Path Router

The path router delegates to two sources for paths: one for the initial path, and another for when rerouting is requested or forced. The initial path may be a fixed route precomputed somehow, but the rerouter must be able to pathfind between any two roads.

By default, to compute a new path to an agent's destination, AORTA uses **congestion routing**, as part of its gridlock avoidance strategy (see §4.3). This is a form of A* [9] that weights each directed road with the pair (congested steps, distance to goal). The first component counts the number of **congested**[22] directed roads in the path so far. Because the ordering is a pair, this means that the routing strictly prefers less congested steps, even if the route is much longer. This preference is quite inflexible and unrealistic: just because a road far away is congested now does not mean it will remain congested once the agent reaches it. The second major application described in this thesis aims generally to improve this.

5.2.3 Drunken Routing

The drunken walk router is not intended for real use. It lazily selects a random turn and lane, caching the answer to be consistent when queried multiple times. Theoretically the agent should eventually reach their goal; in practice, this will not happen. One slight variation of this router prefers turns to roads closer to the goal by Euclidean distance. A further variation remembers previous roads traversed and prefers unexplored roads.

5.3 Agent Behavior

The most intricate module is what controls agent **behavior**. During the planning phase of each tick, every agent must set a target acceleration for

[22] A congested road has over 80% of its max capacity filled

the next tick and optionally request lane-changing. Though this interface is simple, the primary implementation is quite complex.

During the planning phase, the agent behavior module technically has access to all information in the simulation. In practice, the view of the world actually used is limited to the following:

- the position and velocity of all agents on the current road and any "upcoming" roads that could be reached during the next tick
- an interface to the router, to know which path to follow
- an interface to request turns from upcoming intersections

This is a reasonable amount of localized knowledge, though routers do access global aggregated information to deal with congestion.

5.3.1 Lookahead Behavior

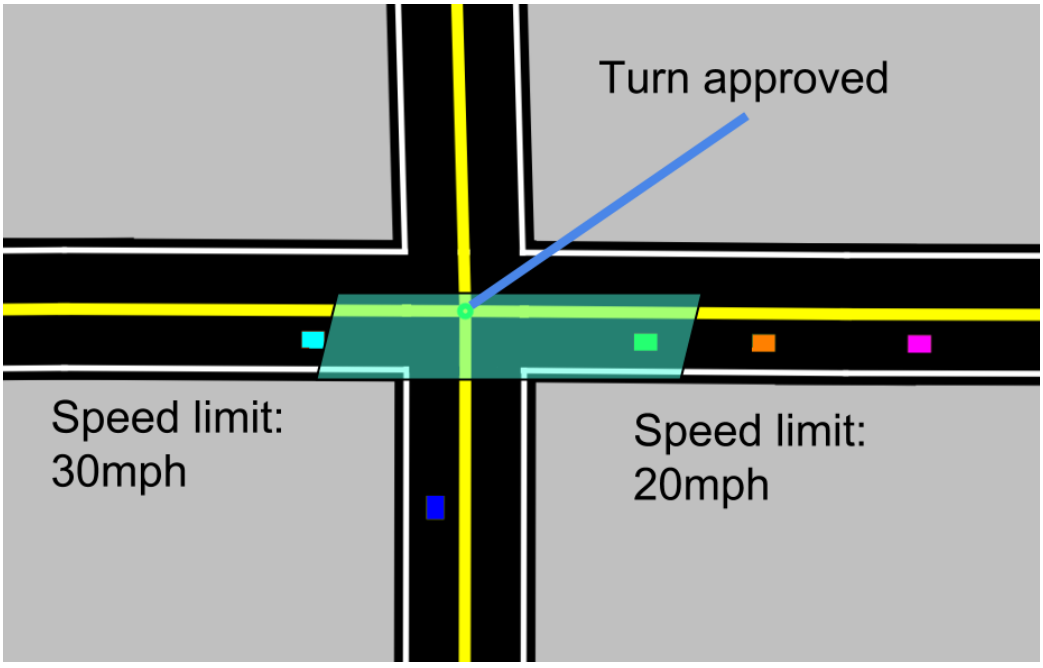

Figure 10: Lookahead and some constraints

The default behavior chooses the highest safe target acceleration by satisfying constraints imposed by other agents, upcoming intersections, and speed limits. Each tick, the behavior computes the max distance its agent could travel during the next tick, then uses that to **lookahead** to the remainder of the current lane and upcoming intersections and lanes, for as long as the distance prescribes. Figure 10 illustrates this: the aqua car scans ahead, verifying its turn is approved at the intersection and paying attention to speed limits and the green car in the next road.

The behavior also has rules for when to attempt lane-changing. If the agent has already requested a turn, then this means that lookahead previously scanned past the end of the current lane and had to request a turn. In this case, the behavior does not try to lane-change, since canceling requests should be avoided. Before deciding to try to lane-change, the behavior performs the same safety checks as listed in §4.1.3. If any check fails, the behavior gives up early, committing to not lane-change on the current road. Agents must commit to remain on a lane before requesting a turn, so this choice favors scheduling a turn sooner. Additionally, when an agent must cross more than one lane in some road, it reduces its speed[23] to increase the chances that this will work. Alternative approaches to lane-changing are possible[24].

The full details of each constraint are available in the code[25], but an overview follows here. First, speed limits impose the simplest constraint. As an agent approaches the end of one road, their lookahead distance causes them to scan to the next intersection and road. The agent will follow the lowest speed limit it observes. The lookahead distance is defined so that the agent will be guaranteed to be traveling at the lower speed limit by the time they reach the slower road.

The second type of constraint is imposed by intersections. An agent may not enter one and perform a turn until approved. Thus, agents must often stop at the end of their lane. Due to floating point issues and AORTA's discretized model of time, there are some cases where the standard kinematic formulas for decelerating to rest while traveling a specific distance fail. For instance, some agents travel at $0.00000001\frac{m}{s}$ for many ticks, slowly reducing their speed to cover a fraction of a meter. In these cases, one fix is to

[23] The agent divides the speed limit by the number of lanes to cross, so more than one lane causes a slow-down. This formula is quite ad-hoc.

[24] Humans tend to slow down if they cannot lane-change due to other cars. This approach has not been explored yet in AORTA.

[25] *utexas/aorta/sim/Behaviors.scala*

accelerate at $a\frac{m}{s^2}$ one tick and then at $-a$ the next, so that the total distance is the desired amount.

The third constraint is to not collide with the **leader** vehicle. In some cases where the intersection is small and the agent is near the end of their road, the leader may be on the next road. Regardless, the worst case that could happen is that during this tick, the leader slams on their brakes, decelerating at the max rate possible. A safe behavior must ensure that during the next tick, the agent can safely decelerate to avoid collision and maintain the minimum required following distance.

5.3.2 Human Behavior

AORTA generally takes the assumption that all vehicles are autonomous and can communicate with upcoming intersections. However, this assumption is not strictly necessary. The lookahead behavior drives as fast as is safely possible[26] and models an AV with excellent reaction time. An easy way to model human drivers (beyond changing their router) would be to use the target acceleration prescribed by the lookahead behavior as a limit. In other words, assign each driver a parameter indicating how responsive they are during the trip, and use it to randomly choose an acceleration some percentage or fixed amount less than the base-line acceleration from lookahead. This has not been implemented.

[26] except for parts that are overly conservative because the math is easier

6 Intersection Auctions

Traditional traffic control schemes assign all drivers the same priority. Drivers, though, have different values of time. A traveler late for a flight presumably tolerates delay less than somebody going shopping. An ideal traffic control system should be able to accommodate this difference. This system could not simply poll drivers for their level of impatience, rewarding some while punishing others, since most drivers would lie. Instead, the drivers must support their priority by spending form of money.

In previous work [3], we proposed and prototyped **intersection auctions**, a way to let drivers express their preferences at intersections. Rather than admit drivers in the traditional first-come, first-serve order, auctions are run to determine the intersection ordering. This section is an updated form of that paper. §6.1 starts by reviewing other systems for regulating traffic with auctions. Then §6.2 formalizes the system for running intersection auctions. Rather than making humans participate in the auctions, "wallet agents" described in §6.3 bid on behalf of drivers. Because introducing auctions raises equity and efficiency concerns, §6.4 regulates the auctions with "system bids" that subsidize bids beneficial to the overall traffic flow. Lastly, §6.5 evaluates the system's performance and §6.6 discusses further implications of it.

6.1 Related Work

A similar approach to our work was conducted by Vasirani and Ossowski, who extended the AIM reservation protocol[7] to use auctions at individual intersections as well. To handle an entire network, they introduce **competitive traffic assignment**[20], which has intersections update reserve prices in real-time in response to the current demand. Drivers choose routes based on their own preferences between time and cost, participating in intersections auctions as long as they are willing to meet the reserve price. Although our work bears similarity to this approach, there are several key distinctions. First, our system does not restrict drivers to routes based on pricing. Instead, we use pricing to control the order in which drivers proceed through intersections. Second, we explicitly address the issue of equity with **system bids**. Third, we generalize the notion of auctions to work at traditional intersection policies as well as with autonomous reservations. Finally, we show that the idea of intersection auctions is compatible with the realism of a microscopic simulator running on maps from OpenStreetMap, and make our

implementation available under an open source license.

Another similar work offers valuation-aware intersections[16]. The auction format differs in that time is a relevant factor in their **free choice** and **clocked** modes. Rather than run an auction as soon as the intersection is ready to accept another driver, their work runs auctions at different times to maximize the number of participants. The evaluation also differs in that we run simulations with our auction protocol at city-scale, while this work focuses solely on a single intersection.

6.2 Auction Framework

This section first presents an example of running an intersection auction, then formalizes the general procedure and describe details of how it works at each type of intersection.

6.2.1 Example

As an example to illustrate how our proposed intersection auctions work, consider Figure 11, in which driver A is about to finish crossing the intersection. The amount each driver bids (in units of cents) is shown below or beside their name, and the total bid for each choice at the front of a queue is circled. All drivers wish to cross straight across the intersection. Only B, C, D, and E are valid items for the auction, since the other drivers are stuck behind one of these four. Each driver bids for the driver at the front of its lane, since each is self-interested. Note that the drivers behind C do not bid anything, while the driver behind E is actually helping E, to get the indirect benefit of moving up in their queue of cars sooner.

The winners will split the cost of the second highest bid with **proportional payment**, a method inspired by the Clarke-Groves tax mechanism[5, 8]. B wins with 50¢, and the runner-up is E with 40¢. The winners – B, who bid 30¢, and G, who bid 20¢, must together pay the total of the runner-up's bid: 40¢. Since B comprised 60% of the winning bid and G made up the other 40%, B pays 60% of the 40¢ and G pays the remaining 16¢. After B crosses the intersection, a new auction is run, and G is now a candidate. Note that G must bid again; the fact that it contributed to the winner of the previous auction does not affect the next auction. In fact, there is a risk that new drivers, not shown, could appear behind C and D and repeatedly

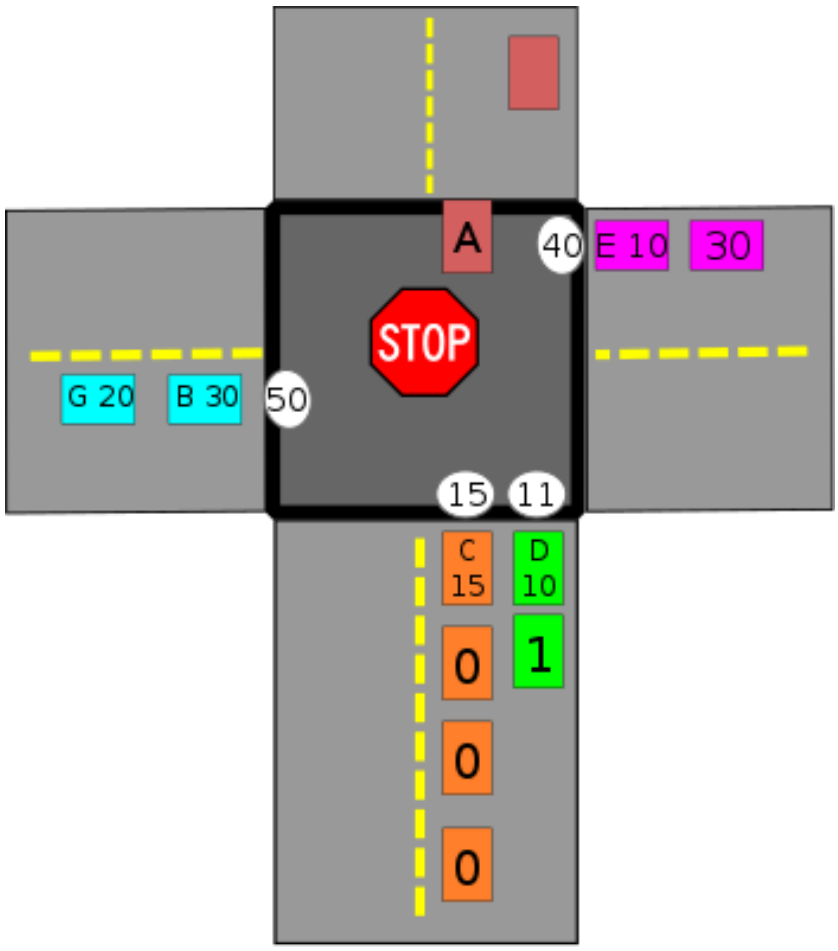

Figure 11: An example of intersection auctions applied to a stop sign

outbid G. The effect is that G wastes its money paying for B to move earlier. This reflects a more general risk, discussed in §6.6.2.

6.2.2 General Procedure

This section formalizes the procedure for running an intersection auction. The **participants** of auctions are all drivers on a road leading to the intersection. These participants choose and bid for various **items**, which are either individual drivers wishing to perform a turn or an entire phase of a traffic signal. (Drivers can bid for multiple items since more than one phase might be useful to a driver at a traffic signal.) After summing all bids for an

item together, a separate **system bid**, denoted $SB(item)$, is multiplied to boost certain items, described later in §6.4. As a result of the auction, there are some **losers**, which are participating drivers who experience delay due to the outcome. The items up for bid and the losers are defined per intersection policy in §6.2.3. The auction process is thus:

1. Ask all participating drivers to bid on an item, collecting *bids*
2. Determine the winning item: $argmax_{item}(SB(item) * sum(bids_{item}))$
3. With the winning item chosen, determine the runner-up item and losers, then the compute the second price: $SB(runner\_up) * sum(\{bid \in bids_{runner\_up} \mid bid_{agent} \in losers$
4. Collect payment from each driver who bid for the winning item. If a contributed *amount* to the *total*, then a pays: $\frac{amount}{total} * \frac{second\_price}{SB(winner)}$

The auction format is that of a 2^{nd} price, sealed bid auction. There is a single round, with drivers oblivious to the bids of others. The winners split the *second_price* cost proportionally to what they originally bid. Dividing by the system bid prevents a driver from spending more than it bid. Note that ties are broken arbitrarily (but deterministically), and do not occur often in practice.

6.2.3 Auctions at each Intersection Policy

Auctions are incorporated in AORTA's three intersection policies, which are defined in §5.1.

For stop signs, the *candidates* that drivers can bid on include all drivers who are stopped at the front of their respective lane. The *losers* are all drivers who did not bid for the winner, which should be everybody in a different lane from the winner, since they do not benefit from the outcome.

For traffic signals, the *candidates* are phases, rather than individual drivers. The winning phase lasts for the usual fixed duration, but agents can extend this duration by bidding for it consecutively. The *losers* are drivers whose desired movement is not in the winning phase. Since a turn could be in multiple phases, it makes sense for a driver to bid for all phases including their turn. Additionally, agents could predict whether or not they could make the next light by counting the cars in front of them.

Recall that reservation managers approve one driver at a time until the approved driver's turn conflicts with other scheduled turns. The *candidates* are all drivers that are not blocked by unaccepted drivers (since the gridlock invariants of §4.3 must be maintained). The *losers* are all drivers whose turn conflicts with that of the winner, since they have no chance of winning the next auction and being immediately accepted.

6.3 Wallets: Automatic Bidding

Bidding at each intersection requires constant attention and unbiased judgments about how to best divide the budget. Humans are ill-suited for both task, so each autonomous vehicle conceptually has a **wallet agent** bidding on behalf of the human. Ideally, humans input their preferences to the wallet in the form of constraints: a max budget to not exceed for this trip, a limit on cost per time gained, and a deadline for arrival. However, enforcing these constraints requires predicting the effects of moving through an intersection several seconds earlier. Due to the sequential auction problem discussed in §6.6.2, for now, flexible wallets must be deferred.

Two simple wallets are currently implemented. The **static** wallet always bids some fixed amount and has an effectively infinite budget. This can be used to model emergency vehicles, for instance, by making the amount that they bid sufficiently high. Alternatively, drivers could pay a one-time fee at the beginning of each trip, and the priority they express at every intersection is this one value.

Second, the **fair** wallet attempts to divide the total budget evenly among all intersections. At any intersection, it spends the remaining funds divided by the number of intersections left. Since routes change when drivers encounter congestion or cannot change lanes, the number of remaining intersections sometimes changes. By default, the fair wallet only pays once at each intersection: it does not pay for anybody ahead of its driver. This has the drawback of preventing drivers with a high budget from achieving time savings when they are stuck behind slower drivers. There is also a bias in the fair wallet of spending more money towards the end of a trip. Since auctions are second-price and the burden of payment is shared among the winners, the fair wallet usually spends less than it bids. As a result, it spends $\frac{1}{3}$, $\frac{1}{2}$, and finally all of these remaining funds at the end of the trip.

6.4 System Bids

This section will describe various issues that may arise when individuals express preferences that lead to inefficient overall orderings. To regulate these issues, an agent separate from all drivers will place **system bids** to maintain various fairness properties. These bids act like a reserve price, meaning drivers can always overcome them by bidding high enough.

There are several types of system bids for different circumstances, and since they sometimes conflict, it is necessary to tune the magnitude of their bids against one another. At the same time, system bids must be scaled against driver bids. If system bids are too low, they can never affect the outcome of auctions. If they are too high, driver preferences will be ignored. The method we use is to *multiply* driver bids by system bids, rather than add system bids. After the bids for some item are summed, the system bids for that item are multiplied by the total. (This is the rate, denoted R, in §6.2.2.) By multiplying instead of adding, drivers can bid between 5¢ and 10¢ or $5 and $10, and the regulation will be the same. (When system bids apply to an item, 1¢ is added, since multiplying by zero has no effect.)

The process of tuning system bids sets the specific multipliers for each type of system bid. Right now, the rates are manually set. In the future, they will be adjusted by hillclimbing or genetic algorithms with an offline test.

In the remainder of this section, specific system bids will be presented.

6.4.1 General System Bids

Figure 12 demonstrates two general problems that may occur when drivers express preferences. On the left side, A wants to move south into a full queue, while B wants to turn left into an empty queue. A clearly will not save time overall by hastening to wait in a different queue, so B should move first. The system bid for **pointless impatience** corrects this by rewarding drivers destined for queues with more capacity available. Specifically, drivers moving into queues with at least %25 max capacity free are rewarded by having their bid multiplied by $capacity\_rate = 5$.

The right side of Figure 12 shows another problem. Driver C wants to turn left into a busy queue that has steady demand from the north. Imagine C is not willing to spend much money, so C must wait indefinitely to make its turn. This situation is particularly troublesome if C has low-income and

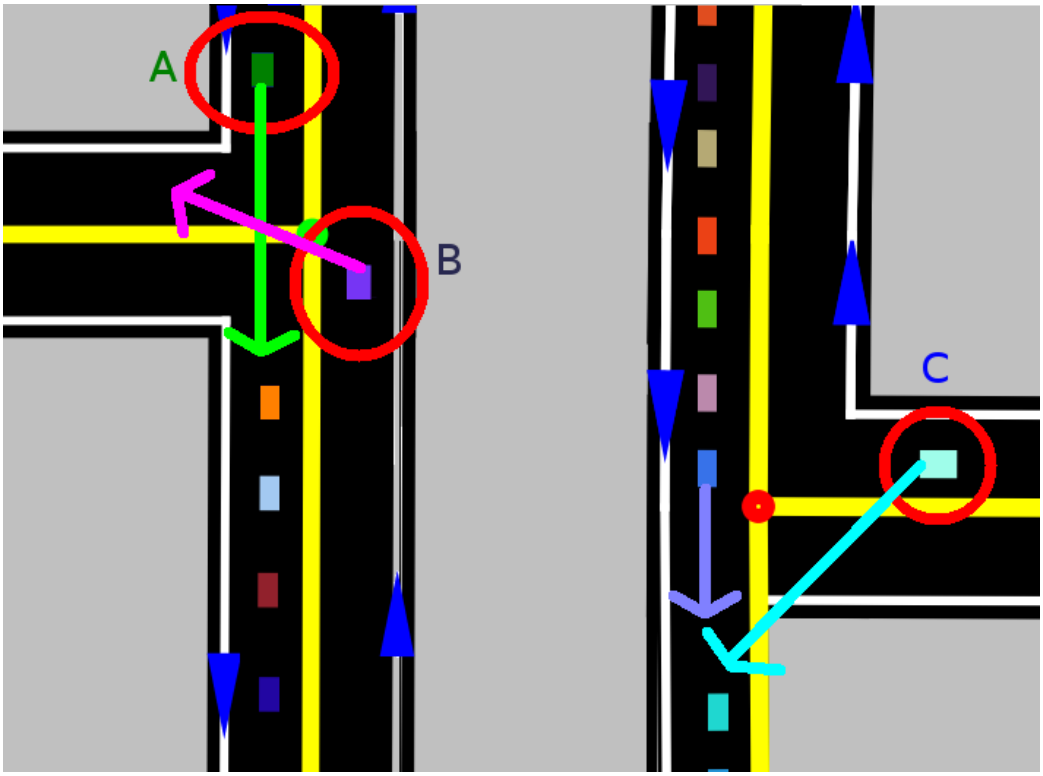

Figure 12: Two scenarios ripe for regulation by system bids

cannot spend much money. Thus, a system bid will help drivers avoid waiting forever. A driver's **waiting time** is the time since the driver stops because it reaches an intersection or the driver in front stops. The system bid for **liveness** multiplies C's bid by $wait\_rate * waiting\_time$, where $wait\_rate = 1$ currently.

A third issue relates to gridlock, which is discussed in §4.3. An empirically effective way to relieve congestion is to reward drivers trying to leave full queues, since other drivers are often blocking their queue by waiting for a full queue. The system bid for **dependency** multiplies bids by $dependency\_rate * demand$, where $dependency\_rate = 2$ currently and $demand$ is the number of drivers in the queue.

6.4.2 System Bids for Reservation Managers

Intersections controlled by reservation managers, described in §5.1.3, admit drivers in some order. When this ordering is poor – if the winning agents occur on alternating directions that conflict – the throughput of the intersection degrades severely to that of a stop sign. System bids can help.

The first system bid relates to which drivers are eligible to win. Drivers request turns before they reach an intersection, but accepting a driver far away from the intersection needlessly prevents conflicting drivers from passing. Stop signs only admit drivers stopped at the intersection, and traffic signals cut off drivers too far away to possibly cross before the phase ends, but reservation managers need the help of a system bid to achieve a similar effect. The driver in the front of a queue is ready to turn if they are close to the end of the queue. Other drivers are ready to turn if they are closely following a driver that is ready. The system bid for **readiness** multiplies bids by ready agents by $ready\_rate = 5$.

To prevent drivers in conflicting directions from winning too often, the system bid for **throughput** multiplies bids by drivers with turns compatible with those previously accepted by $throughput\_bonus = 7$. Note that throughput is not applicable for stop signs and is naturally achieved by traffic signals.

6.5 Experimental Evaluation

This section evaluates the performance of intersection auctions using two metrics. First, **unweighted total trip time** is the sum of every agent's trip time[27]: $\sum_{agent} time_{agent}$. Lowering unweighted time means agents on average experience faster trips. Second, **weighted total trip time** is a sum of trip time, weighted by the budget each driver has available during the trip: $\sum_{agent} time_{agent} * budget_{agent}$. Minimizing this metric involves lowering the trip time of drivers with larger budgets, capturing the intuition that higher-paying drivers should experience better trips. Of course, comparing the weighted time of two individual drivers is meaningless, as the score is not normalized to account for the route length.

Seven different **modes** will be evaluated in this section. **First-come, first-serve (FCFS)** represents the status-quo, where drivers cannot express

[27] trip time begins when an agent attempts to spawn, meaning some time is spent waiting for a safe moment to appear on a busy road

their priority at intersections. In **equal** mode, all drivers use a static wallet, always paying $1. This treats all drivers the same, but intersections make choices based on unweighted demand. For instance, at stop signs, the driver in the most congested lane will win. Next, **fixed** mode assigns all drivers the static wallet with a uniformly distributed budget. At every intersection, the driver bids this amount. This scheme models drivers paying a one-time fee for their trip and experiencing that priority at each auction. Finally, drivers in **auction** mode pay using a fair wallet and a uniformly distributed initial budget. Equal, fixed, and auction mode – all except FCFS – can work with or without the aid of system bids, with the default untuned configuration presented in §6.4.

Although FCFS is the status quo, using it as a baseline to benchmark other modes may be inappropriate. All other modes are aware of the number of cars on each incoming lanes, while FCFS at traffic signals will blindly cycle through a phase that nobody uses. Additionally, the reservation manager with FCFS ordering admits drivers in a nearly arbitrary order and may degrade to the performance of a stop sign if turns often conflict in the ordering. A better baseline is equal mode with system bids, which still ignores the priority of drivers, but has knowledge of demand.

6.5.1 Single Intersection Test

First, we wish to quantify the effects of auctions at a single intersection. The setup consists of one reservation manager-controlled intersection, with 4 roads connecting, each with 2 lanes in each direction. Vehicles spawn for one hour with budgets uniformly distributed between $1.00 and $2.00. Two modes are compared: the baseline FCFS and auctions. For auctions, two factors are further varied – whether or not system bids are used, and whether the agent's fair wallet bids for agents ahead in the same queue.

Figure 13 shows the results of using system bids at various levels of vehicles spawned. The y-axis is unweighted time savings, which is more positive when the auction mode performs better than FCFS. As the coloring shows, system bids are necessary for auctions to be useful in this setup. Consider how the reservation policy without system bids operates. Since budgets and agent locations are uniformly distributed, the winner of an auction is from an arbitrary road, with a %50 chance of conflicting with the previous winner. The system bid for throughput addresses this.

Figure 14 shows results only when system bids are employed, coloring

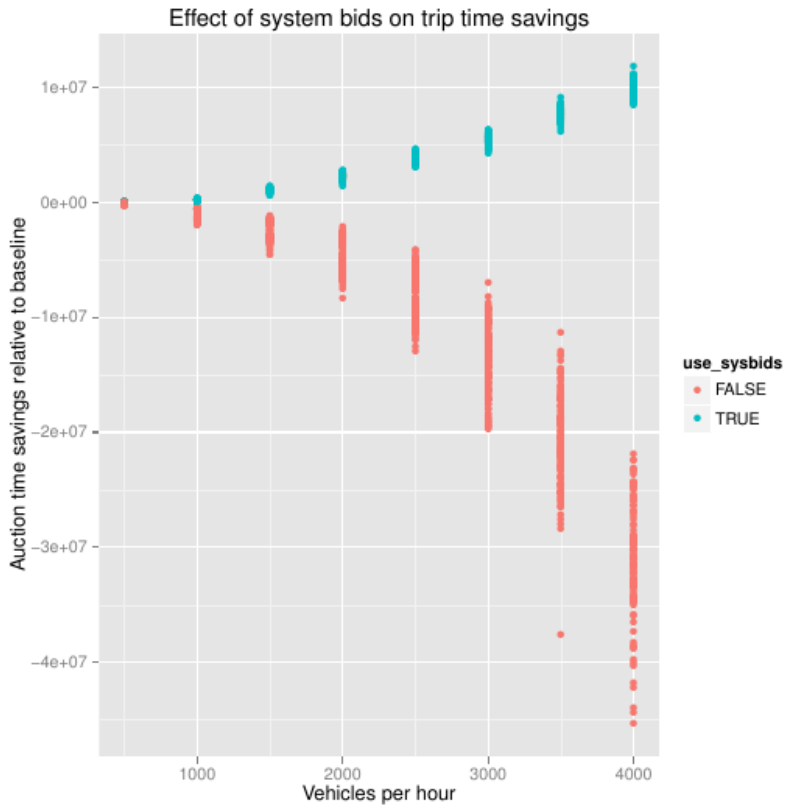

Figure 13: Effect of system bids on trip time savings at one intersection

based on whether or not agents bid for drivers ahead. Based on these results, this factor is not an indicator of unweighted trip time performance.

Although both unweighted and weighted time were measured, the correlation coefficient between the two metrics over all runs is 0.999845, meaning total weighted time is effectively the same as unweighted with re-scaling. This means the auction mechanism did not serve drivers with higher budgets better than drivers with low budgets – all drivers experienced faster trips, but higher-paying drivers were not preferred.

6.5.2 City-wide Tests

This section evaluates the 7 modes in 4 cities: Austin, Baton Rouge, San Francisco, and Seattle. Between 10,000 and 15,000 drivers spawn per hour

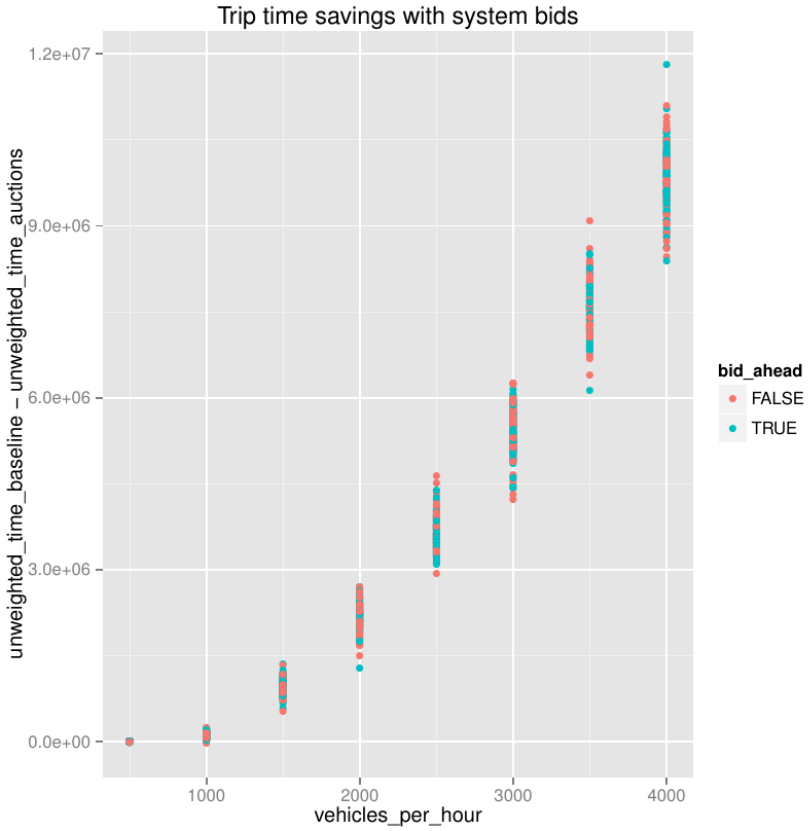

Figure 14: Trip time savings with system bids at one intersection

for 3 consecutive hours, then the simulation proceeds without spawning additional drivers until all drivers complete their trip, or a deadline of 12 hours is reached (indicating some failure like gridlock or a lane-changing bug). Runs with up to 30,000 drivers per hour have been successfully completed, but this scale takes a prohibitively long time to simulate. Drivers have uniformly chosen sources and destinations[28] and a budget uniformly distributed between \$0 and \$5.00. Agents route with shortest-distance routing initially and reroute away from congestion.

During a test run with the same parameters, agents used the fair wallet but did not bid for agents ahead in the queue at stop signs or reservation managers. Figure 15 shows the distribution of how much of the total bud-

[28]This distribution is unrealistic, but generating more natural demand distributions is difficult without trip data

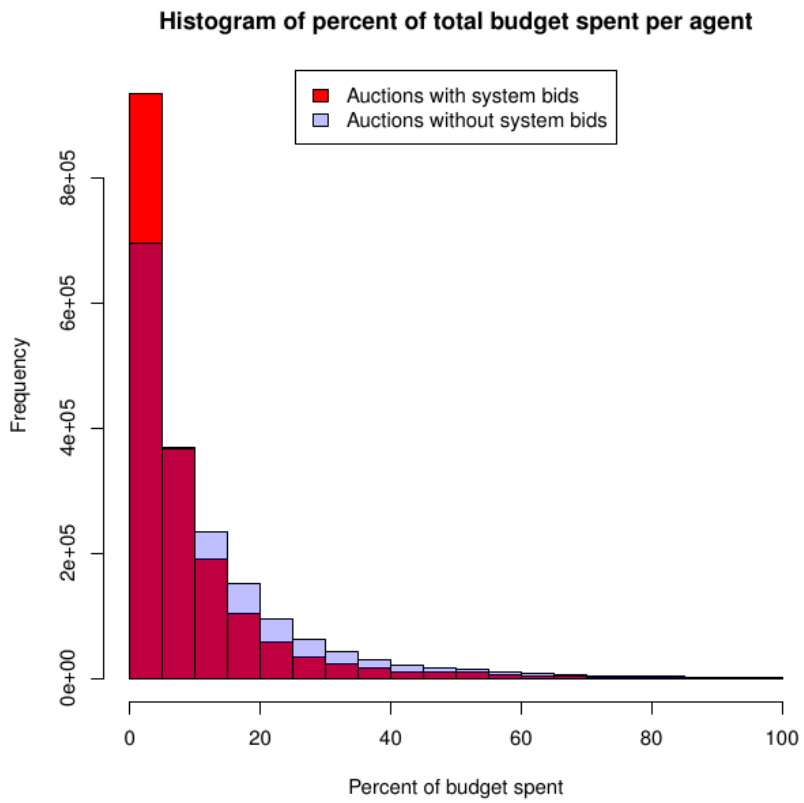

Figure 15: Percent of budget spent with fair wallet when bidding ahead is disabled

get each agent spent. The effects are more pronounced when system bids subsidize part of the cost. Of course, bidding ahead is not the only factor contributing to the percentage of budget spent. This preliminary run simply motivates bidding ahead to be enabled in the final tests presented next.

Figure 16 shows the unweighted normalized trip times from different scenarios in each city. For each scenario, the trip time of every agent is summed up, producing one total time per scenario. To account for the different number of agents in each scenario, this time is then divided by the number of agents. The raw trip time is given in seconds, but the graphs show a more understandable scale with minutes. Note the number of trials in Austin, $n = 3$, is low due to bugs[29]

[29]Austin has some short roads with many lanes, causing agents to attempt impossible lane-changing forever.

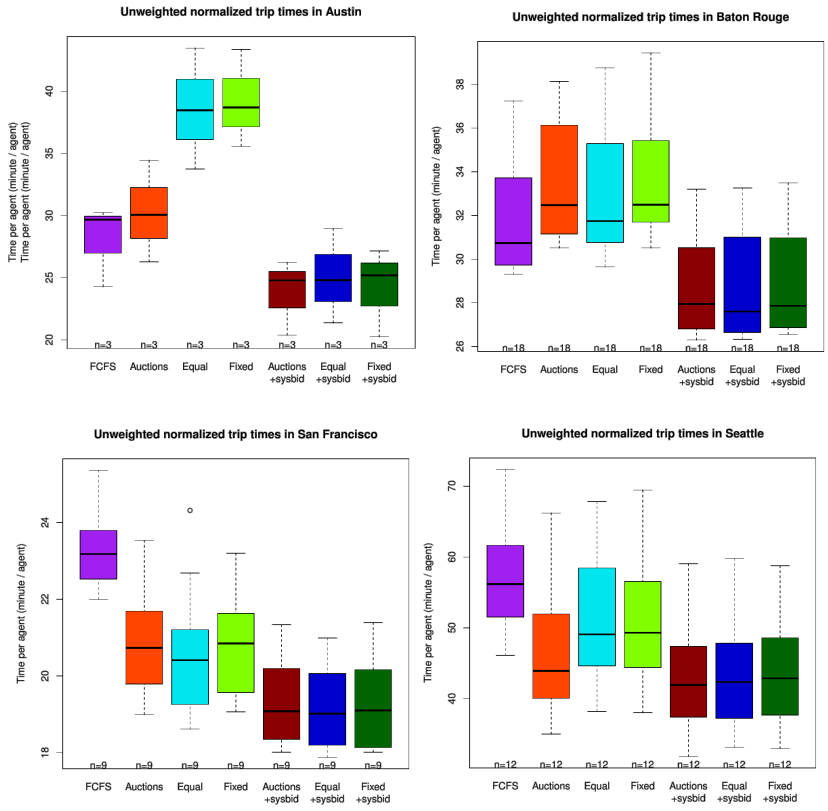

Figure 16: Unweighted trip times in 4 cities

To compare modes more easily, we show unweighted trip time savings relative to two baselines: FCFS and equal with system bids. Figure 17 shows the savings relative to FCFS. In Austin and Baton Rouge, auction, equal, and fixed mode without system bids perform *worse* than FCFS – up to a 15 minutes loss per agent. In San Francisco and Seattle, FCFS truly is the worst mode. In all cases, it is clear that auction, equal, and fixed mode perform the best.

FCFS is a strawman; the true baseline is equal mode with system bids. Figure 18 shows the unweighted time savings relative to equal mode. In Austin and Seattle, auction and fixed mode perform marginally better than equal. In Baton Rouge and San Francisco, they perform marginally worse. Under the conditions, this suggests that the unintentional primary contribution of intersection auctions is the use of system bids. They improve the

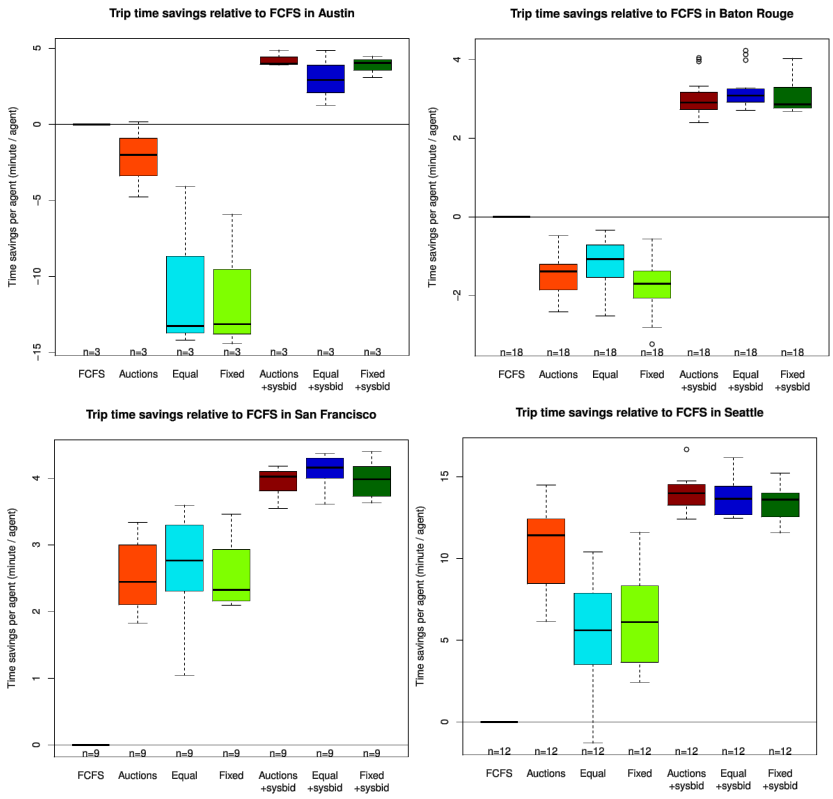

Figure 17: Time savings relative to FCFS

time of every agent without regards to their level of priority. Further experiments are required to isolate which system bid contributes most to savings. Note that in auction mode, every agent used the same strategy for bidding – the fair wallet – and this does not represent a true market. More bidding strategies should be simultaneously employed.

Additionally, more data suggests that during auction mode, agents were not fully utilizing the auction system. Figure 19 shows the percent of the total budget that each agent spent. Note this figure is different from Figure 15, since agents bid for drivers ahead of them in the final run. Although this distribution has more mass at higher percentages, few agents spent more than %40 of their budget. Either agents did not have enough opportunities to actually participate in auctions, or the fair wallet does not spend aggressively enough. It is future work to evaluate both possibilities. The results of this low spending are shown in Figure 20. Just as in the single intersection test,

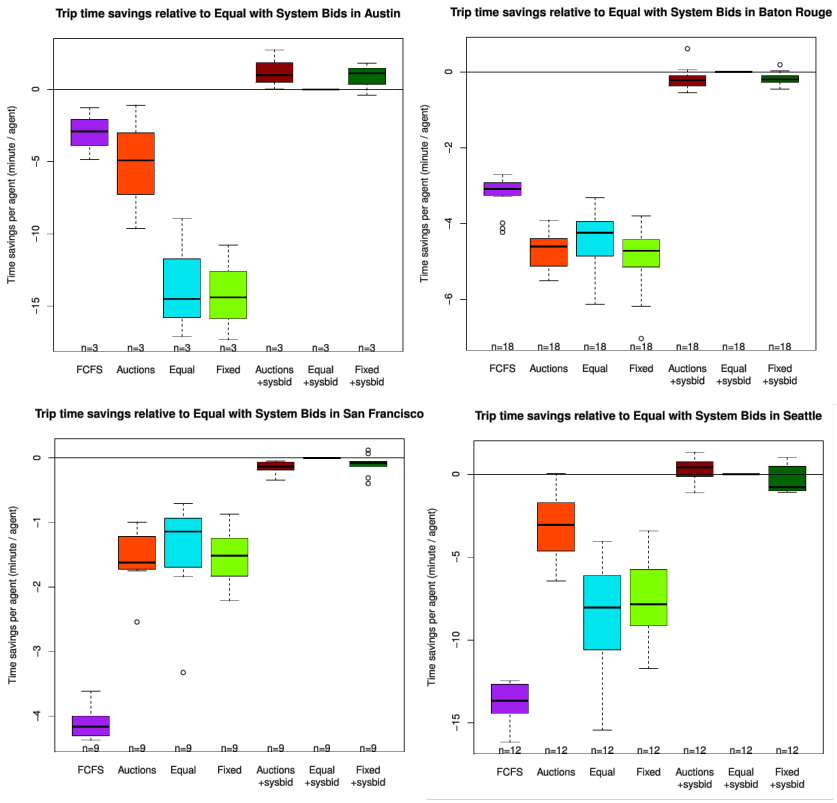

Figure 18: Time savings relative to Equal mode with system bids

the unweighted and weighted time correlate nearly perfectly. Higher-income agents did not spend much more, so their time savings did not improve relative to lower-income agents.

6.6 Discussion

This section raises some conceptual issues with our approach. First there is the issue of how to use the money collected during intersection auctions. One idea is to replace road maintenance taxes with the earnings. Rather than have everybody pay equal taxes for a common good, drivers who receive preferential treatment could pay more. However, this could force agents to strategize by taking into account their utility for supporting road maintenance. The remainder of this section presents other concerns.

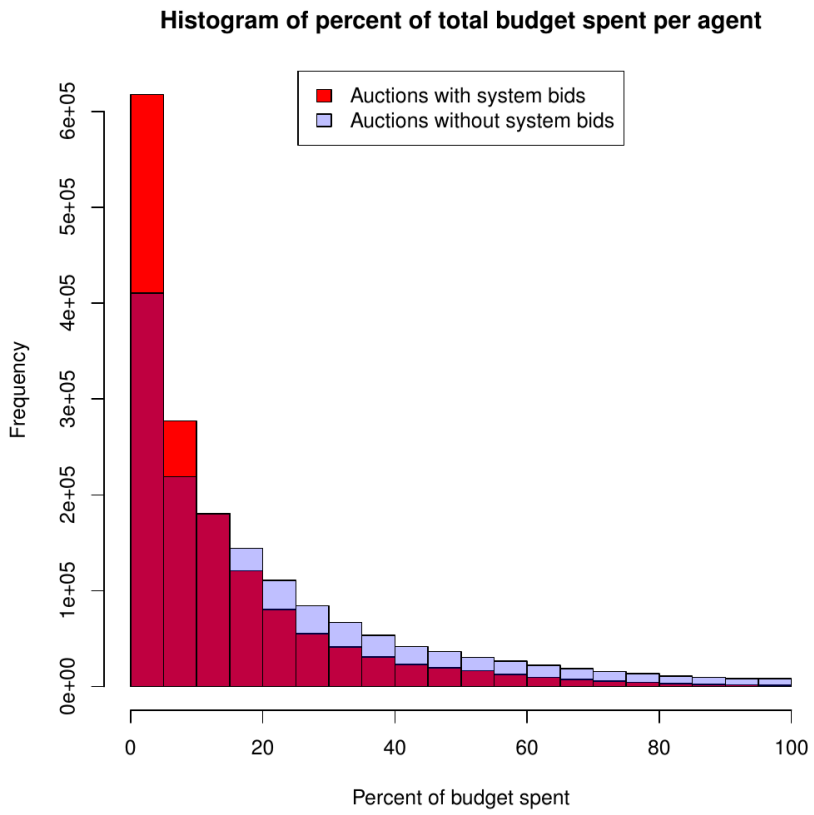

Figure 19: Percent of budget spent with fair wallet when bidding ahead is enabled

6.6.1 Exploitation

Suppose some malicious agent E drives slowly. Drivers stuck behind E may consider helping E at intersections, to make up lost time. To remove this incentive for E, any reasonable wallet implementation could explicitly detect bad drivers and never reward them. If this policy is universal, then E has no reason to delay itself and others.

Another potential concern is that most drivers would free-ride instead of using the auction system. However, people have a non-negligible value of time. If nobody bids, the cost to improve time is low. Additionally, second price payment means that drivers who bid against a would-be free-rider will get priority but pay nothing, encouraging participation.

The system should protect drivers with lower budgets from having to wait indefinitely. The system bid for waiting time directly addresses this

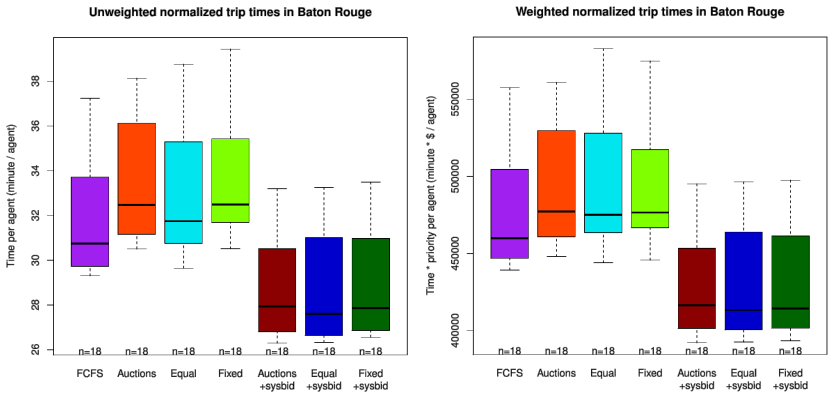

Figure 20: Unweighted and weighted trip times in Baton Rouge

danger. Another option is to grant each driver a fixed number of "credits" to everybody to use instead of money. These credits would be renewed per week or month, and drivers could spend them however they see fit – all at once for one fast trip, or spread out carefully. To be fair, credits should be tracked per driver, not per car, since wealthy drivers could afford multiple cars to get more credits.

6.6.2 Sequential Auctions

Suppose some driver is willing to pay up to $10 to complete a trip by some deadline. After carefully spending $8, they are close to reaching their destination. Unfortunately, a traffic jam forms around their destination, and either the $2 remaining is insufficient to reach the goal before the deadline, or the traffic jam is so severe that no amount of reordering turns could let the driver finish in time. The driver effectively wasted their $8. The driver may be able to spend much less money and still get approximately the same trip time in the end. The fear that this may happen could lead some drivers to spend no money at all.

This problem also exists in the current state of driving, though. Human drivers often leave for a destination early in order to make a deadline. When they do this, they accept the the risk that a traffic jam will cause them to miss the deadline anyway. They may prefer to leave later after the jam clears, still miss the deadline, but spend less time driving.

If trip time cannot be accurately predicted, one alternative to this issue is

time insurance. A driver pays an up-front fee, then engages in intersection auctions as usual. If they bid reasonably but are thwarted by unforeseen congestion, then the insurance company would either refund the bids they spent or help with the remaining bids. The driver takes the usual risk of insurance fees: if no problems occur during the trip, then the fee is the additional, unnecessary cost.

6.6.3 Alternatives to Auctions

This section briefly outlines alternate schemes for optimizing weighted trip time. One possibility is that when a driver D wants to turn quickly, D could compensate any drivers they delay. The delayed drivers could **bargain** with D to set a "bribery" price, which D would pay to the drivers. However, this creates an incentive for somebody to drive in busy areas, yielding to impatient drivers and collecting money. If this money balances out the price of gas and the value of time, bargaining could actually *increase* the congestion in that area! This idea has been implemented in the literature with **time-slot exchange**[17], but the game theoretic downsides are not discussed.

A second method, which supplements intersection auctions, is to change routing to distribute drivers more evenly through roads leading to their destination. These routes could be planned in advance, with shorter routes being reserved for higher-paying agents. A fundamental question with this approach is how to handle planning routes for new drivers. If an agent enters the system later than others, has a route that could cause delays for drivers whose schedules and routes have already been set, and is willing to pay more than the others, how should the existing routes change?

The third alternative, placing tolls on roads to prevent drivers from crowding a popular route, is the subject of the remainder of this thesis, starting in §7.

7 Externality Pricing

Although drivers have many possible paths to their destination, they often choose a familiar path, deviating when they sense higher traffic levels directly or via GPS. When drivers do plan routes, they often ignore their own impact on traffic. A direct route through the busy center of a city may be faster than a lengthy but quiet detour, but it could also cause marginal delays on other drivers in that congested region. Taxing routes through congested regions may encourage traffic to distribute more evenly. Along with predictions of the time to follow a route, agents could evaluate potential routes in terms of a time and cost trade-off.

§7.1 starts by quantifying the effects of a route on other drivers. §7.2 then presents an initial experiment based on the simplest externality model. Next, §7.3 shows a simple and effective method for predicting trip time. Finally, §7.4 concludes with ideas for extending the pricing scheme.

7.1 Externality

There are many notions of externality, such as fuel emissions, noise pollution, and creating traffic in residential neighborhoods. Here we focus just on time externalities. Suppose in world A, a population P of drivers perform some trips. In a derivative world B, one new driver D is present. Then one way to measure the externality imposed by D is to sum the difference in trip time for all the other drivers:

$$externality = \sum_{driver \in P} time_B(driver) - time_A(driver) \tag{1}$$

If some driver finishes their trip before the new D joins or is traveling far from D, then their time in both worlds is likely to remain exactly the same. D can certainly influence drivers with whom it has no direct contact: if it causes a slight delay at an intersection, everybody moving through that intersection in the future will be impacted slightly. This makes this definition extremely prone to noise[30].

An alternative is to blame the new driver not for every propagating effect they unknowingly cause, but instead for a more easily measured, localized effect. Consider a popular road R. It has a **freeflow capacity**, the max

[30]This has been observed in attempts to measure this form of externality in simulations.

number of cars that can be on the road simultaneously such that all of the cars can travel at the max speed limit. R also has a **rest capacity** – the most cars that can co-exist when all the cars are stalled. When R approaches its freeflow capacity, many new drivers should not choose to take R, since they will slow themselves and others down. At the same time, R should strive for max utilization, if it could contribute to shortest routes. One way to limit the number of drivers that include R in their route and mediate this instance of the tragedy of the commons is with a **toll**. In this section we evaluate a simple linear toll: $toll = \max(0, percent - 50)$, where *percent* is the percent of freeflow capacity filled. Roads under 50% freeflow capacity are free, then the price increases linearly. Note *percent* can exceed 100%, since rest capacity is the true limit for a road.

7.2 Preliminary Experiment

With the toll primitive, several modes respecting tolls can be evaluated. All the modes route using A* with a 2-tuple scoring each step. Steps are ordered lexicographically, meaning lower values in the first component are strictly preferred and the second component is ignored until two steps have the same value in the first. For all modes in this experiment, the secondary component is a loose measure of route time – distance divided by speed limit[31]. The **avoid max** mode tracks the highest toll of any road in the current path so far. The **avoid sum** instead adds the toll at each step, preferring routes that go through roads with the least total toll cost. Finally, as a **baseline**, congestion routing is used, which has the number of congested roads as a cost. Congestion in this router is defined as more than 80% of its total capacity filled, while tolls rise as freeflow capacity exceeds a threshold.

These 3 modes are tested in 4 cities, with between $10,000$ and $15,000$ drivers spawning per hour for 3 consecutive hours. All intersections use auction ordering with the help of system bids, with every agent having the same *equal* fixed budget. (As §6.5.2 shows, this mode performs much better than FCFS.) Note that in this experiment, agent do not pay tolls. Nor do agent preferences affect their routes at all. Ideally, agent value of time can be used to assign higher-paying drivers to in-demand, fast routes and lower-paying drivers to less frequented, perhaps longer routes. For this initial

[31]It would be quite reasonable to use a more accurate time prediction, such as the one developed in §7.3 instead.

experiment, all agents will avoid any sign of congestion.

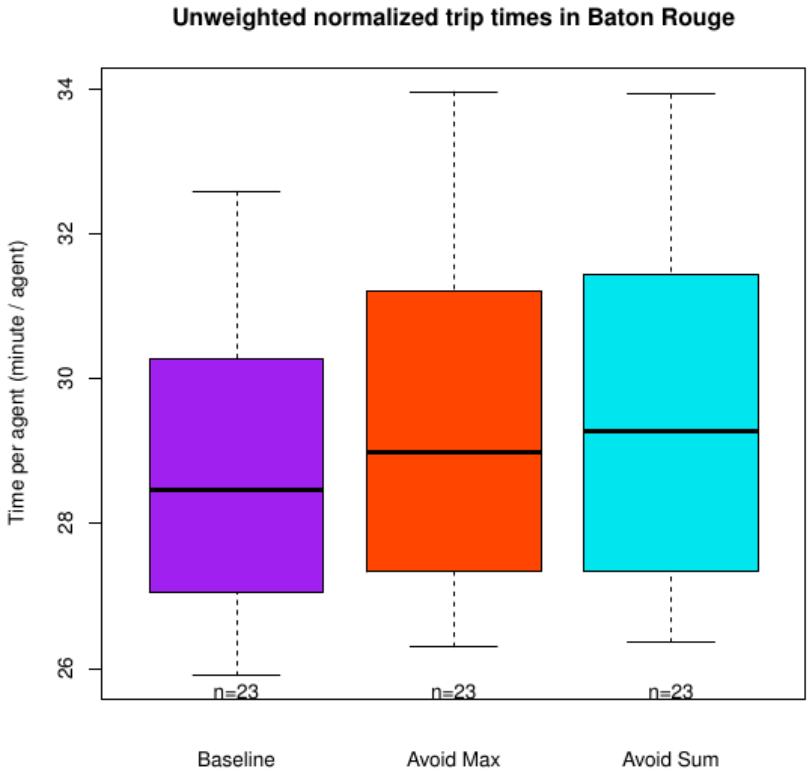

Figure 21: Normalized trip times per mode in Baton Rouge

The results surprisingly differ significantly by city. In Baton Rouge (Figure 21) and Seattle (Figure 22), routing involving tolls resulted in an average *delay* from the baseline of 30 seconds per agent and nearly 10 minutes per agent, respectively! In Baton Rouge, the source of the problem is not clear: Figure 23 shows a histogram of the freeflow capacity percentages witnessed as agents entered queues. Percentages under 100% are clipped out because they are so much more common than others, making behavior at high congestion harder to discriminate on the scale. In baseline mode, agents observe high congestion much more frequently than when avoiding the sum or max toll. Figure 24 shows the congestion in Seattle. Here, avoiding high tolls actually causes *more* agents to enter highly congested queues, which explains the extremely high delay.

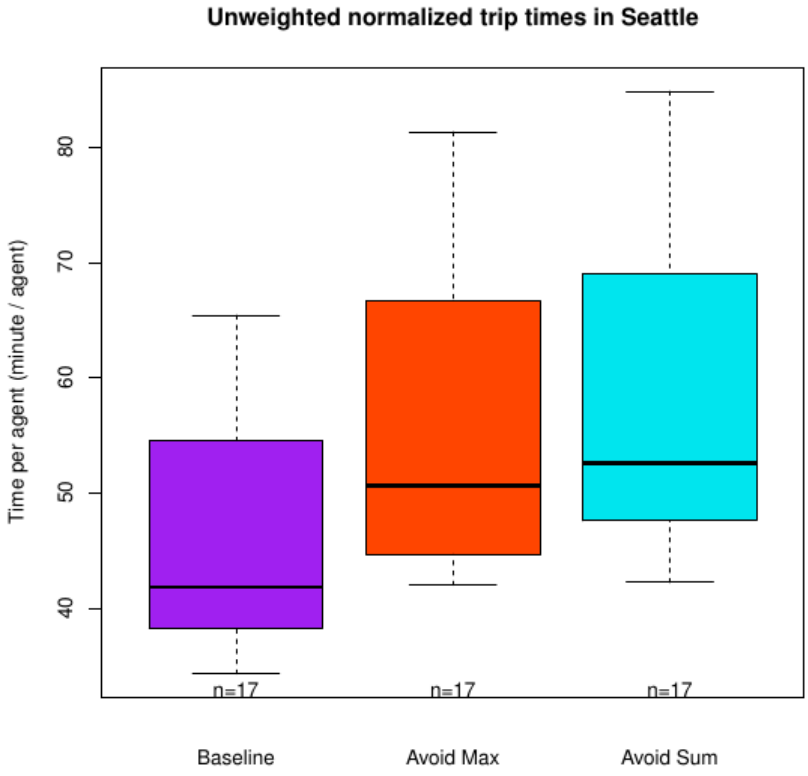

Figure 22: Normalized trip times per mode in Seattle

In Austin (Figure 25) and San Francisco (Figure 26), cars avoiding tolls have on average a gain in time of about 2 minutes per driver. This is a much more encouraging result. The distributions of freeflow capacities observed in both cities are much more similar to the distribution in Baton Rouge. As Figure 27 and Figure 28 show, baseline mode causes agents to enter more congested queues than the other modes.

What factors account for this discrepancy? One possibility is that since *all* drivers avoid roads approaching congested levels, too many drivers needlessly pick longer routes, when some percentage of them could have taken the more direct route. This may be the case in Baton Rouge, where tolls helped drivers avoid congested roads, yet agents had worse times on average. The more likely explanation is that the current condition of a road is not always a good predictor of the road in the future. When roads change state between

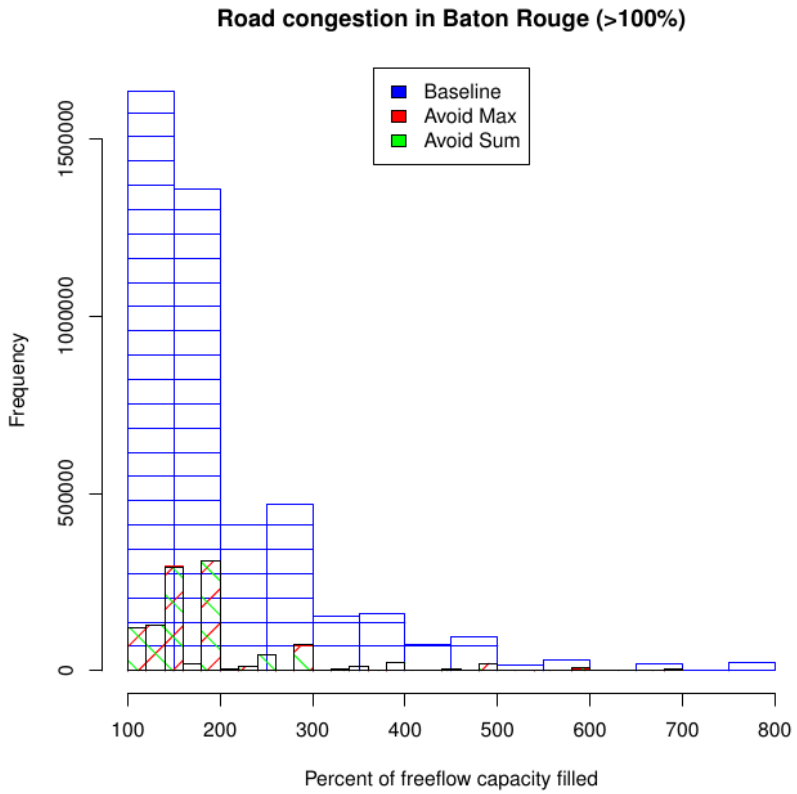

Figure 23: Frequency of entering highly congested roads in Baton Rouge

clear and congested, drivers who have already performed pathfinding will not notice, unless they reroute due to gridlock or failed lane-changing. For some unknown reason, this occurred much more often in Baton Rouge and Seattle.

7.3 Trip Time Prediction

To evaluate trade-offs between time and cost, agents must be able to predict the time it will take to follow a route. The freeflow time (distance divided by speed limit) used in §7.2 is not sufficient. In this section, we present a simple linear model for trip time. The inputs to the model are certain **route features**, all of which are easily measured once at the time that the model runs. A linear combination of these features then produces a predicted time. To learn the weights for this combination, the features of routes and resulting trip times are recorded in a number of trials, and then a linear regression gives

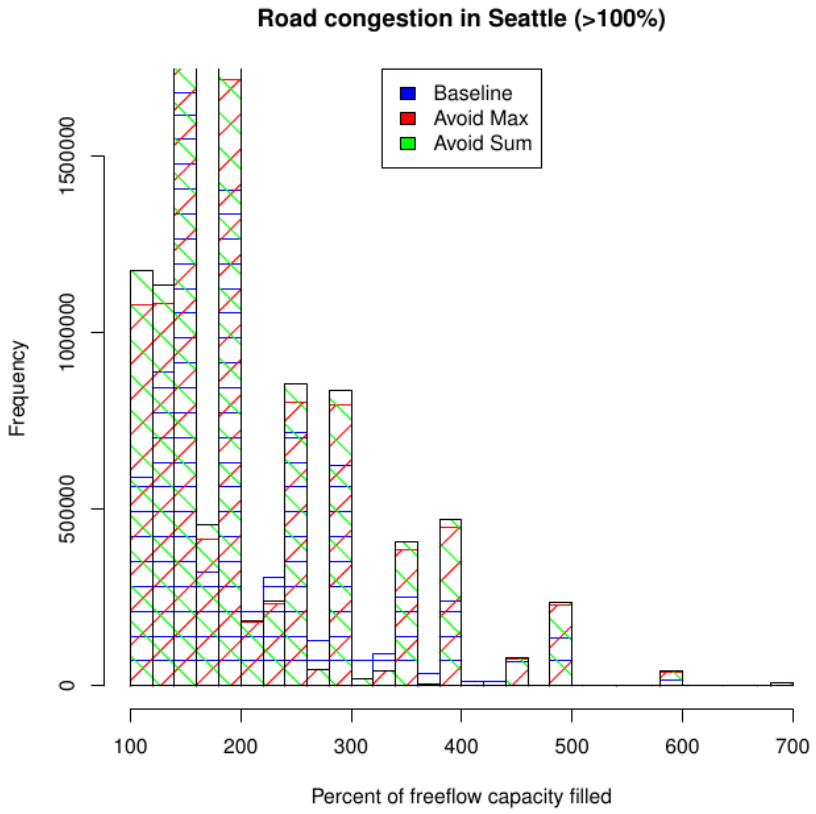

Figure 24: Frequency of entering highly congested roads in Seattle

the weights.

7.3.1 Route Features

This section describes a vector of route features. The dot product of this vector with a vector of learned weights then predicts the trip time. Each directed road along a route contributes one of these feature vectors. The component-wise sum describes the whole route. Note that each road contributes monotonically to the route vector; no negative values occur at any step.

Table 1 describes the 11 features. Note that **static features** hold the same throughout an entire simulation, while **dynamic features** change depending on the time during simulation they are evaluated. Note that calculating road and intersection demand seems to require the path every agent

Table 1: Route Features

	Feature	Description
Static features	length	the length, in meters, of each step
	freeflow time	the length of each step divided by the road's speed limit
	stop sign count	the number of stop signs along the path
	traffic signal count	the number of traffic signals along the path
	reservation count	the number of reservation managers along the path
	road demand	the number of agents who want to use each road sometime
	intersection demand	the number of agents who want to use each intersection sometime
Dynamic features	congested road count	the number of roads currently congested
	queued turn count	the number of turns queued at each intersection in the route
	average waiting time	the sum of the average waiting time for turns at each intersection
	agents enroute	the number of agents currently on each step of the path

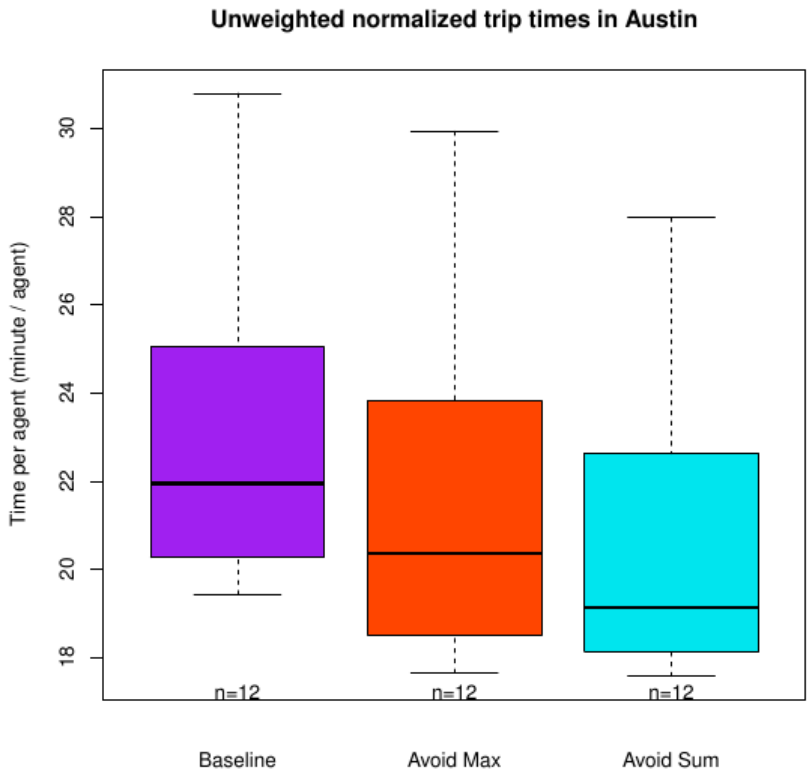

Figure 25: Normalized trip times per mode in Austin

is destined to take, before they choose it. This seems quite unrealistic, but demand simply represents a general notion of which roads and intersections tend to be popular in a city. The dynamic features do not account for agents that will spawn soon, so this demand instead accounts for it. Ideally it comes from historical usage data about the most utilized roads and intersections. In our implementation, the demand does not represent the precise path that each agent will take (since that is impossible to know without simulating); instead, it uses standard shortest-distance routing.

7.3.2 Initial Results

This section presents a learned linear model for trip time. Note that it was generated from an older version of AORTA, and it has not been evaluated against the latest version, which has fixed some bugs. Waikato Environment

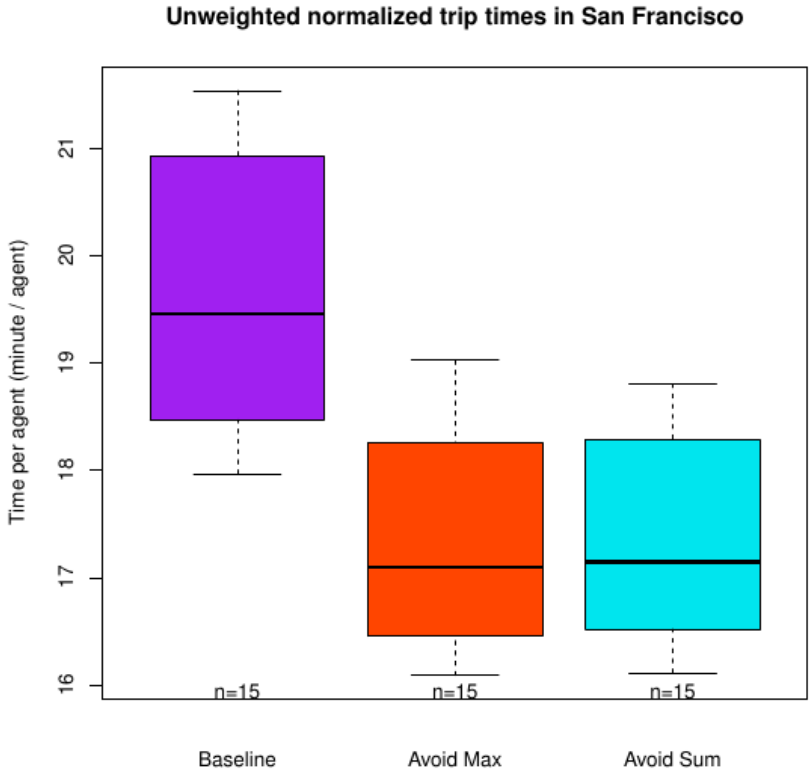

Figure 26: Normalized trip times per mode in San Francisco

for Knowledge Analysis (Weka) was used for all analysis. To generate the training data of route feature vectors and trip times, between $10,000$ and $15,000$ new drivers were spawned per hour for 3 hours, with a source and destination chosen uniformly from the entire map. Simulations were run in 4 different cities, but different numbers of simulation in each city were run[32].

City	Total number of routes recorded
Austin	$105,810$
Baton Rouge	$584,418$
San Francisco	$864,870$
Seattle	$304,488$

Figure 29 shows the distribution of each route feature and the trip time metric. This histogram is generated from combining all the data from every

[32]The map was independently and randomly chosen on many machines in parallel

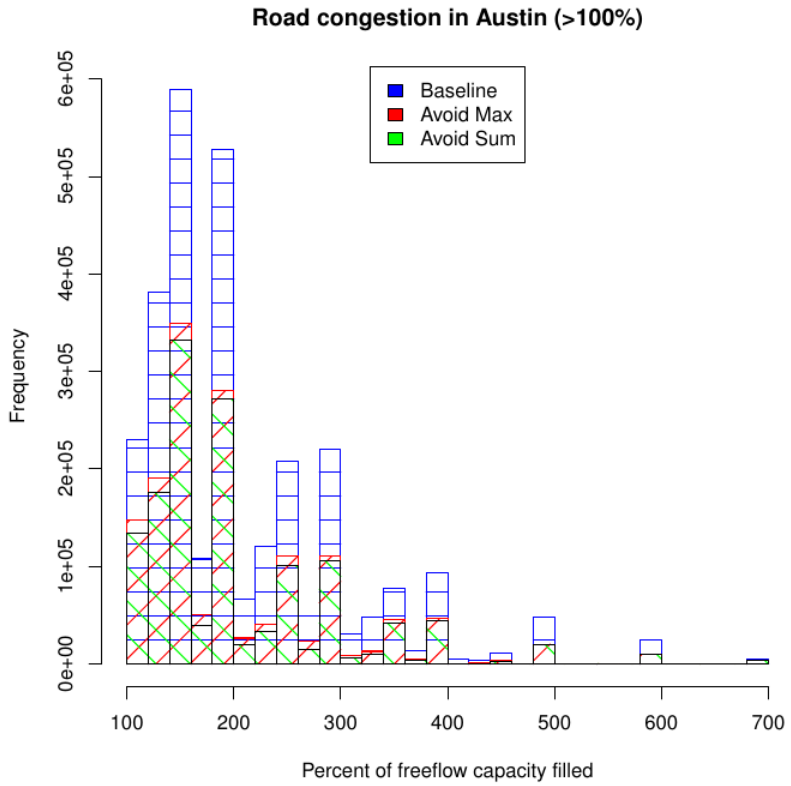

Figure 27: Frequency of entering highly congested roads in Austin

city, but the histogram for each individual city's data looks similar. Most routes do not incur a high penalty for any feature, but a few agents wind up stuck in a traffic jam, constantly rerouting and increasing all components of their route.

Table 2 shows the results of generating a linear model for trip time from all routes from each city, then evaluating against each test city. In all cases, Pearson's correlation coefficient is reasonably high. As expected, the model generated from a particular city is the best at predicting that city. Since the coefficient is so similar across each row, though, this indicates that the model from one city transfers to another. This means that the route characteristics do not omit some factor intrinsic to each map.

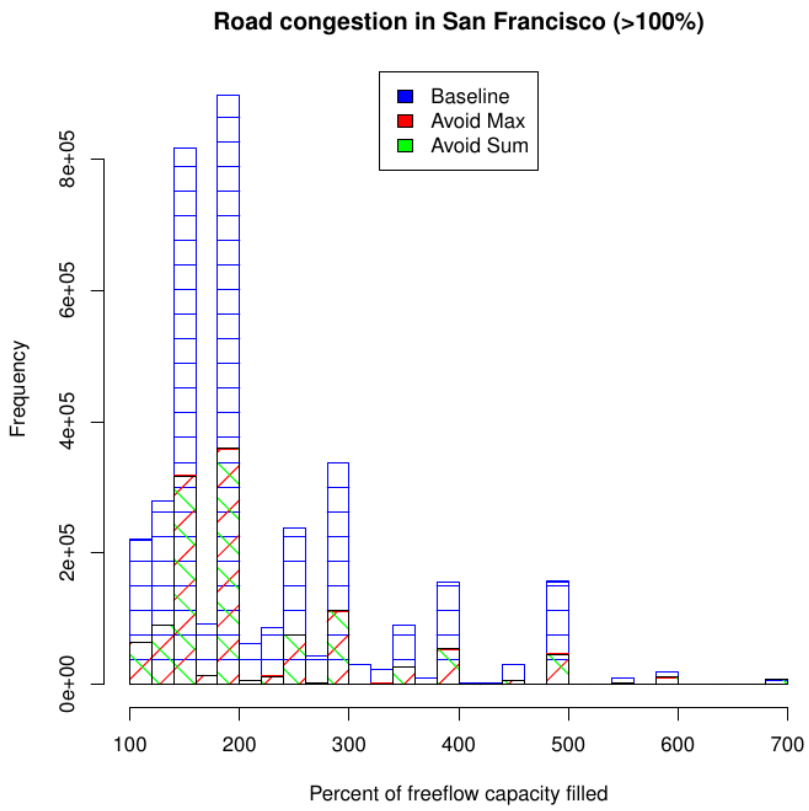

Figure 28: Frequency of entering highly congested roads in San Francisco

7.4 Next Steps

The preliminary results in §7.2 are promising – in some cities, limiting the number of drivers on popular roads leads to overall improvement for everybody. However, the methods so far have not take the agent's own value of time into account. The tolls should encourage the appropriate number of drivers to take popular roads, not reject everybody once half the capacity is reached. One task will be guaranteeing that agents who take a popular shortcut pay an amount that somehow compensates the drivers banned from using that faster route.

The original idea of externality pricing was to present drivers with several choices of routes, each with a predicted trip time and cost. Given those choices, a procedure to choose a route from choices R and a VOT would be as follows:

Table 2: Correlation coefficient of linear model for trip time

		Test city			
		Austin	Baton Rouge	San Francisco	Seattle
Training city	Austin	.8803	.8134	.8343	.8378
	Baton Rouge	.872	.8635	.8435	.8272
	San Francisco	.8756	.8533	.8517	.8329
	Seattle	.8681	.7828	.8129	.8488

Table 3: Algorithm for choosing a route

$baseline = \arg\min_{i \in R}(price_i)$	Choose the cheapest route as the baseline
$C = \{i \in R \mid time_i \leq time_{baseline}\}$	Do not pay more for slower routes
$ratio(i) = \frac{time_{baseline} - time_i}{price_i - price_{baseline}}$	Judge each route by the time saved and additional price, relative to baseline
$r = \arg\max_{i \in C}(ratio(i))$	Choose the best time savings per cost trade-off
Choose r if $ratio(j) \geq VOT$, or $baseline$ otherwise	Do not spend more than the VOT limit
Pay $price_r - price_{baseline}$	The cheapest route is always free

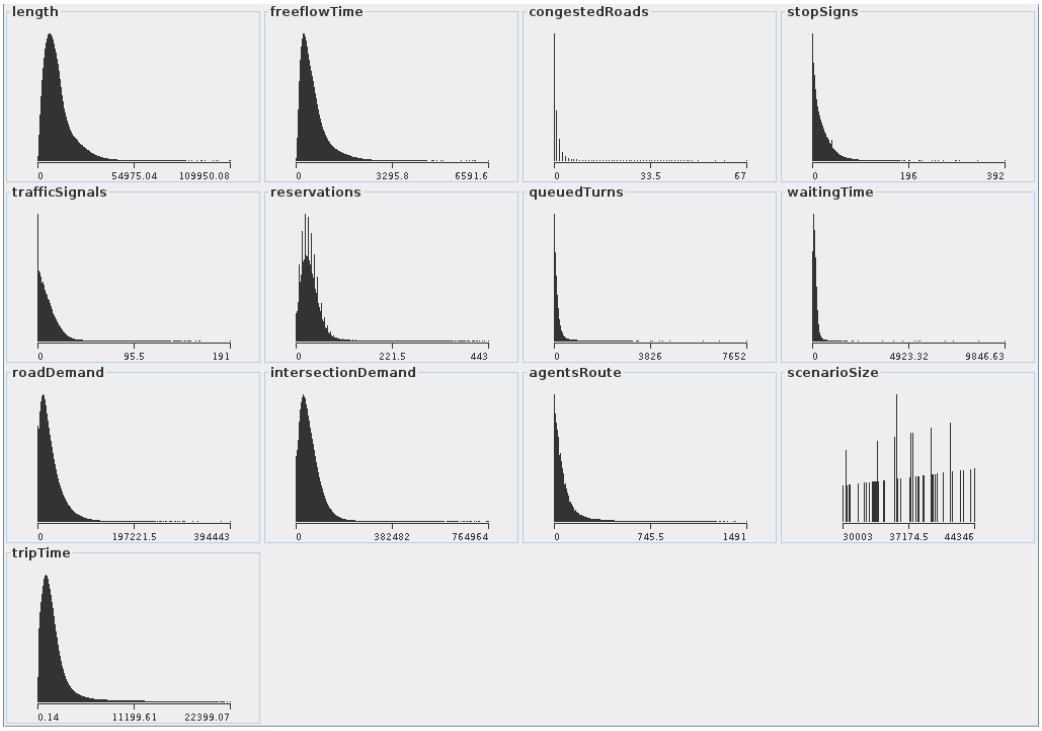

Figure 29: The distribution of each route feature, from all cities' data combined

One way to use this procedure would be to generate several choices of routes for every agent by A* with a step cost based on route features. As the search expands, A* could maintain the total feature vector for the path so far. A step would be scored using the dot product of this vector with a fixed weight vector. Different weights yield different paths. A variety of useful weight vectors could be learned offline. This method, though, is slow to simulate, since many paths for every agent would have to be computed. A faster alternative would be to run A* just once, tracking the time-cost trade-off at every step. A driver would try to avoid roads whose cost is not worth the predicted time savings. The problem here is that time savings require a baseline path of a lower (or free) cost. The baseline would have to give a predicted time and cost for *every* step that A* considers.

In addition to these issues, two assumptions taken in the preliminary experiment could be relaxed. First, drivers currently reroute only when forced to by missed lane-changing or gridlock levels of congestion. In the future,

agents could reroute every few steps, which lets them take into account current conditions. This, of course, would be very expensive to simulate. Second, the toll for roads is currently a fixed, simple linear formula. If the toll's behavior does not match the distribution of driver's values of time, then too many or too few people might utilize a road. It may be wise to adjust tolls online with a reinforcement learning approach.

8 Future Work

This section describes a number of project ideas that could be explored using AORTA.

8.1 Contraflow

Contraflow lane reversal, as the name suggests, reverses the direction of some lanes, adding capacity to one direction of a road. It is currently used for evacuation scenarios and in some regions with predictable morning and afternoon rush patterns. Some challenges to wider deployment involve safety concerns regarding how the infrastructure informs humans about the change. Autonomous vehicles, of course, could respond much more quickly. There has been successful work to reverse lanes dynamically based on current congestion, rather than follow a set schedule, with improvements up to 70% [11].

AORTA could be used to evaluate the effects of contraflow policies in a microscopic setting. Although maps in AORTA are generally immutable, lanes could have a small amount of mutable state describing their orientation. The procedure to determine how lanes are linked by turns could be directly re-used.

8.2 Simulator Performance

AORTA's performance has not been extensively benchmarked, but loosely speaking, there is room for improvement in the number of agents that can be simulated at a time. Aside from profiling and optimizing bottlenecks, there are two broad strategies for improvement.

8.2.1 Fast-paths

Recall that discrete event simulators jump forwards variable amounts of time to the next interesting event. One idea to speed up AORTA is to do the same on a per-agent basis. There are two **steady-states** in which an agent is not changing their behavior. First, when an agent is traveling at the full speed limit, they will remain at freeflow as long as they are far from the next intersection and the leader vehicle, if any, is doing the same. Second, when an agent is at rest waiting for an intersection, either at the front of

their queue or behind another stopped driver, they remain stalled until the intersection state changes. In both of these cases, there is no need to run the full lookahead behavior from §5.3.1, which consumes about half the time by very rough profiling. Instead, the agent can run a **fast-path**, which would quickly return a default action.

Only certain events trigger an agent to enter or leave steady-state. These have not been analyzed in detail yet, so they may not be sufficient or necessary. To enter the freeflow steady-state, an agent must be traveling at the speed limit. They must leave when the leader vehicle exits the freeflow state, or when they get within some distance of an intersection. To enter the stalled steady-state, an agent should be at rest. They must resume normal lookahead behavior when their leader vehicle exits the stalled steady-state, or when the intersection notifies them. Both cases are a simple recursive formulation, which could also be implemented quite efficiently. To test that fast-paths function correctly, it would be necessary to run many simulations, verifying there is no difference between in output from runs using fast-paths to normal runs.

8.2.2 Parallel AORTA

At some point, only so many agents can be simulated quickly on one machine. Using multiple processors in a single node or even a cluster of nodes could allow more agents to be simulated concurrently. The expensive part of each tick, the planning phase, is not quite parallelizable (see §4.2.1 for details). Rather than making each agent be a unit of parallelism, it may make sense to group agents on some region of the map together in a node. This may require some sort of load-balancing, in case one physical region of the map has many agents, while another has few. Each region could be simulated independently, but any drivers near the borders would have to be synchronized. One option is to run in lock-step, having each node simulate one tick in their region, then sharing any information about agents that have crossed the border. This approach would be bounded by the slowest machine, but good load-balancing could implement a form of work-stealing to help weaker nodes. Alternatively, nodes could optimistically simulate several ticks before synchronizing, an approach explored before in traffic models [24]. If a node failed to introduce a new agent at the border, it could rewind and try again. A third option seems to be the most lucrative: nodes managing adjacent regions could each cover a small chunk of neighboring regions. In other words, some areas would

be simulated by multiple machines. If the overlapping area is, for example, a set of long roads connecting two regions, then the minimum time it takes for a driver to cross from one region to another would be known. This way, the nodes only need to synchronize when a driver from their exclusive region enters the overlapping region.

8.3 Reversible Simulation

In a reversible simulator, time could efficiently flow forwards *or* backwards. One possible use of this feature would be evaluating short-term alternatives. If an agent is considering a choice with effects quickly measurable, the simulator could run time forwards, score the choice, then back up to the moment of the choice, and try more branches. This could also be done with savestates, though, and as §4.4.4 explains, saving and loading a snapshot generally takes under a second. A second use of reversible simulation could be for debugging. Suppose the user notices a jam forming. They could mark cars involved with the problem, then reverse the simulation, tracking the potential causes. Then one of the culprits could be forced to reroute or make another choice, and the simulation could run forwards, evaluating whether or not the jam still forms.

A trivial way to implement reversible simulation is by save-stating very frequently and rewinding by loading a previous savestate and simulating forward a few ticks. This takes lots of space, though. In AORTA, agents and various components track certain amounts of state. Reversible simulation would have to reduce this, remembering only key data such as the route followed thus far. Since AORTA is deterministic when simulating normally, perhaps there could be a way to minimize the number of possible predecessor states as well.

8.4 Big Breaux – a Centralized Oracle

Most traffic control schemes take the realistic assumption that they cannot command drivers, merely sway them with economic incentives. Big Breaux[33] is an authoritarian, centralized planner that rejects such a premise. All cars who wish to travel anytime throughout the next day must request their origin, destination, and time of departure in advance. Big Breaux will not only

[33]Rumor has it, Orwell liked Cajun food.

schedule a route for that driver to follow, it will also fiddle with predestination paradoxes by prescribing a precise space-time schedule for the driver. In other words, Big Breaux determines the acceleration that the driver should apply every tick in order to be in the right place at the right time.

In some sense, Big Breaux is like AIM [7], except that it schedules a driver's entire movement, rather than just one intersection. Big Breaux could utilize AORTA as follows. It would simulate the first driver to leave in the day in an empty, desolate world, recording its location at every moment in a time-table. The second driver would then run through an empty world containing one "ghost" – that of the first driver. Its lookahead behavior (recall §5.3.1) would have to obey additional constraints that guarantee the **dystopian invariant**: previously scheduled drivers should not have to change anything about their fate. In other words, the new driver would have to avoid interacting with the first in any noticeable way. This could mean avoiding being a leader car of the first on any queue, or choosing a route that would pass through intersections at different times.

Big Breaux need not be constrained by 21^{st} century ideas like memory limitations or needing to plan for a buffer around vehicles in case they fail to follow their precise plan. In addition, drivers who do not know precisely when they want to travel in advance simply do not travel. In the cold distant future, there shall be no dynamic re-planning of existing routes. Furthermore, any quibbles over the order in which potentially conflicting drivers are scheduled shall be settled by the Ministry of Equity (Newspeak: Miniquit).

9 Conclusion

This thesis has presented the architecture and details of a new agent-based traffic simulator. Prototypes of two traffic control schemes – intersection auctions and congestion-based tolls – have been evaluated. Per original goals, both experiments are reproducible in any city with minimal setup. AORTA's simplicity, stemming partly from the use of a modern programming language, makes it a viable candidate for testing future traffic control schemes.

A Experiment framework

During the course of this thesis, an experiment framework in AORTA has evolved. The goals for it are automation – so repeating experiments with fixed code is trivial – and reusability – so a new experiment can be set up with minimal work. Running an experiment involves three general steps:

1. Write the experiment code declaratively with **metrics** and **modes**
2. Deploy AORTA to a cloud computing environment and run many trials in parallel
3. Generate a report with aggregates from the raw results

To best illustrate the first step, the code for the intersection auction experiment from §6.5.2 is reproduced below:

```
class AuctionExperiment(config: ExpConfig) extends SmartExperiment(config) {
  override def get_metrics(info: MetricInfo) = List(
    new TripTimeMetric(info), new OriginalRouteMetric(info),
    new TurnDelayMetric(info), new TurnCompetitionMetric(info)
  )

  def run() {
    // The generated scenario is the baseline.
    val sysbid_base = AuctionExperiment.enable_auctions(scenario)
    val nosys_base = AuctionExperiment.disable_sysbids(sysbid_base)

    output_data(List(
      run_trial(scenario, "fcfs"),
      run_trial(sysbid_base, "auctions_sysbids"),
      run_trial(nosys_base, "auctions_no_sysbids"),
      run_trial(AuctionExperiment.equal_budgets(sysbid_base), "equal_sysbids"),
      run_trial(AuctionExperiment.equal_budgets(nosys_base), "equal_no_sysbids"),
      run_trial(AuctionExperiment.fixed_budgets(sysbid_base), "fixed_sysbids"),
      run_trial(AuctionExperiment.fixed_budgets(nosys_base), "fixed_no_sysbids")
    ), scenario)
  }
}
```

An experiment changes independent variables and collects data on dependent variables. The independent variables are called **modes**, and they are expressed as transformations on a base scenario. The same experimental setup is precisely reproduced, with only the inclusion of system bids, the distribution of budgets, or the style of payment varied. The dependent variables are the metrics listed. These use the listener pattern to echo raw data such as agent trip time or delays experienced at intersections to plain-text files. Originally, only higher-level metrics – such as the time savings between auction mode and the baseline, or the average of all turn delays – were saved. However, we found that other analysis was desirable several days after the test had run. Rather than re-run the test every time, we shifted to a two-step approach, outputting raw data, and then transforming it to produce a report offline.

Gathering sufficient data is only possible by running many trials simultaneously. AORTA includes simple scripts to instrument deployment to Google Compute Engine[34], an infrastructure-as-a-service provider. Virtual machines may be spun up, running standard Linux images augmented with a startup script to download and set up AORTA. Simulation status and results are retrieved via Google Cloud Storage.

The R statistical computing environment is used to transform raw results into aggregates offline. Marshalling data to and from a transient SQLite database has been found useful for reshaping data – SQL queries are often a natural way to express aggregate functions. The scripts to generate the plots in this document are available[35].

[34]For full disclosure, I have previously interned on the team that develops GCE.
[35]See *misc/auction_report.r*

B Project History

A brief history of AORTA follows. My interest in traffic control began when I was a junior in high school, burning my hours at unsynced red lights in Baton Rouge. At the time, I was building a computer game and was happily employing techniques such as collaborative diffusion[15] for making monsters intelligently surround the player. At some point, the two ideas met, and I thought it would be nifty to have a centralized system route vehicles.

Spring 2011, I joined UT Austin's Autonomous Vehicles FRI stream, where I met Marvin, UT's AV, and learned about AIM[7]. The winter break before the semester started, I had discovered OpenStreetMap. Again something clicked, and my course project became the first Java iteration of AORTA: a program to mangle OSM maps into something interesting, display them, and make cars with the phenomenal ability to pass through one another on the same lane without a lawsuit. I believe at the time the AORTA acronym stood for something much less sensible.

Summers always meant internships and fall 2011 was a fun diversion into the realm of pointer analysis research, but by spring 2012, I had rewritten AORTA in Scala, a much more enjoyable language to use. Rewriting entire code-bases was something I loved doing in my early game engine development days. In this case, it was not a direct port, but instead a chance to re-architect everything. At this point, agents did ridiculous things like request and receive approval for turns in the same tick[36] and cause safety checks to explode every time they landed on a road from OSM with a length of $2.7E^{-11}$ meters.

Fall 2012 is a blur of trying to finally add lane-changing support and haggling with gridlock. Dr. Stone tasked me with tolling every road in a city. That winter, my rubber duck and partner-in-thoughtcrime got me in an economic mood, and by January, the ideas for intersection auctions were flooding out. The toll roads, I was convinced, could not work. Spring 2013 was probably my most active semester.

Fall 2013 has been an adventure as well. Writing this thesis has taken much of my time, but new ideas about routing have intruded. I was foolish enough to try to write my third conference paper in about a week while I was at a conference in an unfamiliar time-zone and couldn't remember that subtraction doesn't commute[37]. In the end, toll roads made more sense than

[36] This was a terrible idea; intersection policies could not see multiple requests at once, and the determinism was brittle

[37] There was a bug in my formula for trip time savings leading me to believe that a new

I thought. As of November 2013, the git repository for AORTA has 825 commits.

I end with a list of milestones and quirky bugs.

- I have made the infamous "epsilon bug" too many times. When $x == 0$ is inappropriate due to floating-point imprecision, $x < \epsilon$ is **not** the right idea.
- It took over a year of development to draw cars as little rectangles rather than little circles.
- Back in the day, to deal with intersections placed too close together, Mike Depinet and I conceived **uber-turns**, a sequence of turns and abysmally short roads in between that should be grouped. The eventual solution in §3.3.3 still falls short of this original vision due to having a boring name.

system was performing terribly, when, in fact, it was performing quite well

References

[1] Faststats accidents or unintentional injuries. Technical report, National Center for Health Statistics, 2012.

[2] Michael Balmer, Marcel Rieser, Konrad Meister, David Charypar, Nicolas Lefebvre, and Kai Nagel. Matsim-t: Architecture and simulation times.

[3] Dustin Carlino, Stephen D. Boyles, and Peter Stone. Auction-based autonomous intersection management. In *Proceedings of the 16th IEEE Intelligent Transportation Systems Conference (ITSC)*, October 2013.

[4] Dustin Carlino, Mike Depinet, Piyush Khandelwal, and Peter Stone. Approximately orchestrated routing and transportation analyzer: Large-scale traffic simulation for autonomous vehicles. In *Proceedings of the 15th IEEE Intelligent Transportation Systems Conference (ITSC 2012)*, September 2012.

[5] Edward H. Clarke. Multipart pricing of public goods. *Public Choice*, 11(1), September 1971.

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

[7] Kurt Dresner and Peter Stone. A multiagent approach to autonomous intersection management. *Journal of Artificial Intelligence Research*, 31:591–656, March 2008.

[8] Theodore Groves. Incentives in teams. *Econometrica*, 41(4):617–31, July 1973.

[9] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107, 1968.

[10] Matthew Hausknecht, Tsz-Chiu Au, and Peter Stone. Autonomous intersection management: Multi-intersection optimization. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, September 2011.

[11] Matthew Hausknecht, Tsz-Chiu Au, Peter Stone, David Fajardo, and Travis Waller. Dynamic lane reversal in traffic management. In *Proceedings of IEEE Intelligent Transportation Systems Conference (ITSC)*, 2011.

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

[13] Sven Maerivoet and Bart De Moor. Cellular automata models of road traffic. *Physics Reports*, 419(1):1–64, 2005.

[14] Michael G McNally. The four step model. 2008.

[15] Alexander Repenning. Collaborative diffusion: Programming antiobjects.

[16] Heiko Schepperle and K Bohm. Auction-based traffic management: Towards effective concurrent utilization of road intersections. In *E-Commerce Technology and the Fifth IEEE Conference on Enterprise Computing, E-Commerce and E-Services, 2008 10th IEEE Conference on*, pages 105–112. IEEE, 2008.

[17] Heiko Schepperle, Klemens Böhm, and Simone Forster. Towards valuation-aware agent-based traffic control. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, page 185. ACM, 2007.

[18] David Schrank, Bill Eisele, and Tim Lomax. 2012 urban mobility report. Technical report, Texas Transportation Institute, 2012.

[19] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[20] Matteo Vasirani and Sascha Ossowski. A market-inspired approach for intersection management in urban road traffic networks. *J. Artif. Intell. Res. (JAIR)*, pages 621–659, 2012.

[21] YUHAO Wang. Synopsis of traffic simulation models. 1996.

[22] Wikipedia. Autonomous car — wikipedia, the free encyclopedia, 2013. [Online; accessed 16-October-2013].

[23] Tichakorn Wongpiromsarn, Tawit Uthaicharoenpong, Yu Wang, Emilio Frazzoli, and Danwei Wang. Distributed traffic signal control for maximum network throughput. *CoRR*, abs/1205.5938, 2012.

[24] Srikanth B Yoginath and Kalyan S Perumalla. Reversible discrete event formulation and optimistic parallel execution of vehicular traffic models. *International Journal of Simulation and Process Modelling*, 5(2):104–119, 2009.