Leveraging Vehicle-to-Infrastructure Communications for Adaptive Traffic

Signaling and Better Energy Utilization

THESIS

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in the Graduate School of The Ohio State University

By

Manas Agrawal

Graduate Program in Computer Science and Engineering

The Ohio State University

2013

Master's Examination Committee:

Professor Bruce W. Weide, Advisor

Professor Paul A. G. Sivilotti
Abstract

According to a recent report by the US Treasury Department, America wastes $8 billion annually in energy costs because of traffic congestion. Adding the cost of lost time, the damage is said to reach around $100 billion. Moreover, high energy consumption adds to air pollution and contributes to the global warming problem. Infrastructure where different entities (cars and traffic signals) can communicate with each other offers the potential for reducing this waste. But by how much? Suppose full information about location, velocity, and acceleration of each vehicle were available for all vehicles in the vicinity of an isolated traffic signal. Could an intelligent traffic signal predict and adjust to the best possible traffic light cycle times to minimize fuel loss per vehicle? If light timing were changed dynamically based on real-time information from new traffic arrivals after a small interval of time, how much lower fuel loss could be achieved than by basing timing on macro-level metrics such as flow rates and limited vehicle information such as that provided by in-pavement loop detectors? Answering these questions involves developing a simulation framework that is based on an understanding of typical yet safe vehicle operation (by human drivers or autonomous vehicles) and of various traffic arrival patterns, as well as the ability to estimate fuel loss (and/or other optimization objectives) in many different situations.
This document is dedicated to my family.
Acknowledgments

I thank my advisor Professor Bruce Weide and my other thesis committee member Professor Paul Sivilotti for the discussions, comments, and insights. I would also like to thank Dr. Ted Pavlic for all the help he has provided.

This work was supported in part by NSF grants CCF-0811737 and CCF-1162331.
Vita

November 1984.................................Born

2008.............................................B.E. CSE, University of Pune

2011 to present ................................Graduate Research Associate, Department
of Computer Science and Engineering,
The Ohio State University

Fields of Study

Major Field: Computer Science and Engineering
Table of Contents

Abstract ................................................................................................................................. ii

Acknowledgments ............................................................................................................... iv

Vita ........................................................................................................................................ v

Fields of Study ..................................................................................................................... v

Table of Contents ................................................................................................................ vi

List of Tables ......................................................................................................................... vii

List of Figures ....................................................................................................................... viii

Chapter 1: Introduction ....................................................................................................... 1

Chapter 2: Related Work ................................................................................................... 9

Chapter 3: Method and Architecture ............................................................................... 15

Chapter 4: Results and Conclusion ................................................................................ 34

References .......................................................................................................................... 42

Appendix A: Code ............................................................................................................... 44

Appendix B: Sample of the log file .................................................................................. 45
List of Tables

Table 1: Equations ................................................................................................................. 26
List of Figures

Figure 1: Effects of interruption on fuel usage (for one vehicle) ........................................ 5
Figure 2: Effects of interruption on distance travelled (for one vehicle) ......................... 6
Figure 3: Interrupted travel time as taken from [4]................................................................. 7
Figure 4: Cars following each other, as explained in [8] (Follower/Followee model)..... 11
Figure 5: Explanation of different terms in the traffic situation ............................................. 16
Figure 6: Explanation of delay parameter..........................................................Error! Bookmark not defined.
Figure 7: Diagram depicting the simulation with lists of cars and lights ......................... 24
Figure 8: Main flow for acceleration calculation steps....................................................... 28
Figure 9: Part A - for GREEN light and for vehicles that have crossed the signal ....... 29
Figure 10: Part B – For vehicles within light-area and Light=RED................................. 30
Figure 11: Part C – For vehicles within light-area and Light=YELLOW ....................... 32
Figure 12: Fuel usage for 50 seconds total cycle time with different red-light timings ... 35
Figure 13: Fuel usage for 60 seconds total cycle time with different red-light timings ... 36
Figure 14: Fuel usage for 50% red light timings for different total cycle timings ......... 38
Figure 15: Fuel usage for different flow rates and red light timings for total cycle time of 50 sec ............................................................................................................................... 39
Chapter 1: Introduction

1.1 Introduction and problem

According to a recent report from the US Treasury Department, Americans waste around $8 billion annually in energy costs because of traffic congestion [1]. In another recent study it is found out that the average American spends 100 hours per year on road for the work commute, which is more than the average vacation time of 80 hours per year [2]. Moreover, adding the cost of lost time, the damage reaches around $120 billion according to a report by the Texas A&M Transportation Institute [3].

In addition to the money that is wasted due to traffic congestion, another major concern is the enormous increase in the emission of various greenhouse gases that can also be attributed to the excessive fuel usage. For a different but related issue, traffic congestion can result in a rise of frustration level of drivers that increases red-light violations and ultimately accidents. Safety is the biggest concern in every traffic controller system. A safe follower/ followee distance is a very important factor for safety, and adaptive cruise control is ensuring that in some vehicles. More information on safety features, included in this study, will be presented in later chapters.
The collective effect of vehicles remaining idle at a cross road signal is much more damaging than it appears if costs of lost man hours and economical loses are also taken into account. Most of these problems are common for fixed timing traffic signals. These “dumb” traffic signals operate with the goal to minimize traffic queues, but normally have limited capability to adapt to changes in traffic. A very simple example is a situation where one has to stop and wait at a signal at 2 AM even if there is no other traffic. At a few non-critical intersections, after midnight signals just show a blinking caution light instead of red/green intervals. This process may avoid unnecessary wait but can lead to accidents.

While thinking about addressing the above problems, we need to consider a near-future mixed-traffic situation as opposed to only human drivers. The traffic can include autonomous, semi-autonomous, and human-driven vehicles. With technological advancements in cyber-physical systems, we foresee a more aware traffic system where vehicles can communicate with their surroundings, thus enabling Vehicle-to-Vehicle (V2V), Infrastructure-to-Vehicle (I2V), and Vehicle to Infrastructure (V2I) communications that can support adaptive traffic signaling. Also, extra safety measures can be assured with these communications as a vehicle can know instantaneous parameters of the vehicle ahead that can govern the safe following distance and velocity of the following vehicle.

We consider a scenario of an isolated intersection (that is not directly affected by other neighboring intersections) where the traffic signal timings are fixed. The
intersection comprises a single lane of traffic in both directions. Because of fixed signal timings, it is possible that many vehicles in one direction (with traffic signal as red) stop or slow-down unnecessarily, and end up consuming more fuel that is attributed to their reacceleration phase. Also, for all vehicles queued up before the red-signal, total fuel usage increases as time progresses. This “avoidable” extra fuel usage motivates us to utilize various technological advancements as mentioned earlier, and propose a better approach to handle this problem.

1.2 Our method

Most existing traffic simulation systems consider macroscopic flow rates while solving traffic problems, as opposed to our method which is microscopic and assumes complete information about velocity, acceleration, braking power, etc., of all relevant vehicles. This can be thought of as an extension to sensor-based traffic signals where various sensors obtain vehicle information and some infrastructure systems can make decisions based on that. V2I communication enables the signal controller to adapt according to current traffic. In addition, V2I communications provide extra safety features for a vehicle that cannot be stopped in time, by increasing effective green/yellow time and thus limiting red light violations or crashes.
1.3 Thesis statement

It is possible to design an adaptive traffic light controller that optimizes fuel usage per car for an isolated intersection by using complete knowledge about each and every vehicle in the vicinity, and that simultaneously provides safety features for the ongoing traffic flow.

1.4 Optimization metric

We chose fuel optimization for this work, though various metrics such as time or distance loss could be optimized as part of overall improvement of traffic. Two factors contribute to such losses: idling while slowing or stopped, and reaccelerating to cruising speed. The importance of the latter to fuel losses is significant and non-linear.

Safety is so important when it comes to any traffic optimization that most safety related studies ignore fuel usage. Technologically it is possible to have traffic infrastructures that can perform real-time analysis of traffic flows and adapt to a different (and better) light switching pattern based on a metric such as fuel, time, or distance loss, while maintaining safety. The selection of an appropriate mixture of objectives is a problem in itself, and it will be interesting to see effects of optimizing one metric over another, but for this study it is out of scope and can be explored as future work. Similarly infrastructures can be intelligent enough to identify a different metric as a basis for light switching for different situations and traffic conditions. For example in the morning office rush most people may only care about reaching the office on time and prefer to
take a route (a freeway perhaps) with less traffic even if it is significantly longer in distance or uses more fuel. Again, this is for future work.

The effect of interruption of free flow on fuel usage and time/distance losses can be observed in the following diagrams. Figure 1 shows the projected loss of fuel due to slowing, stopping, and reaccelerating. In our approach we try to minimize the difference in fuel usage between an ideal scenario (without interruptions during travel) and a scenario with interruptions. Similarly loss of distance due to interruption during travel is shown in Figure 2.

Figure 1: Effects of interruption on fuel usage (for one vehicle)
To travel the same distance, with interruption, more time is required. This extra time requirement can be considered as unnecessary time spent on the road. Over time it is possible to have a queue of vehicles waiting at a red light. For this scenario, delay in travel time is shown in Figure 3, taken from [4].

Figure 2: Effects of interruption on distance travelled (for one vehicle)
Figure 3: Interrupted travel time as taken from [4].
1.5 Existing methods

Unrelated to our approach, civil and road transportation engineers address the problem of high density traffic by suggesting various types of interfaces such as cloverleaf interchange or overpasses, but these are limited by cost, space availability, and traffic density. Over time it may be difficult to change infrastructure completely without upsetting the traffic. There are various artificial intelligence agent based traffic controllers that can change light timings to avoid congestion [4] [5]. Traditionally (and appropriately) in most of the traffic algorithms and methods that are used, safety always wins over any optimization parameter, our work is no exception. More detail about other existing methods is given in Chapter 2.

1.6 About this document

This document is organized into four chapters. Chapter 2 focuses on traffic modeling, traffic simulation, and related methods. It also compares this work with existing methods. Chapter 3 describes our method in detail and the architecture of the simulation software and also analyzes the results of the simulations. Chapter 4 concludes and also links possible future studies with current findings. Code is provided in the appendix.
Chapter 2: Related Work

2.1 Traffic Modeling

As a research area, traffic modeling has been widely explored since the 1920s [10]. Over the years, the number of vehicles has gone up significantly and with relatively limited infrastructure improvements this has resulted in traffic congestion problems. Most of the research in this field seeks to minimize congestion and provide efficient traffic movement. It is also a multidisciplinary field of research with civil engineers, computer scientists, city planners, and mathematicians among others working together or lending and sharing their expertise to different parts of the problem. In Chapter 1, we briefly talked about civil engineering and city planning perspectives to solve traffic problems where mostly infrastructure is modified and enhanced. As computer scientists, we approach this problem with modeling, and run and test accurate simulations of traffic systems. This modeling approach also helps in evaluating a potential new infrastructure change (V2I communications) and its effects on traffic. There are many interesting and different directions in traffic modeling; however, for the scope of this work, we focus on the simulation approach.
2.2 Macroscopic Models

Macroscopic traffic models are analogous to fluid flow models. These models attempt to analyze average behavior of the system by noting that traffic flow is similar to the flow of liquid in a pipe [12]. Any individual vehicle behavior is ignored here. In macroscopic models, each road or junction has a capacity. Based on this capacity of the system and average speed of vehicles, an equation to solve (and optimize) traffic problems can be formulated. There are various advantages and disadvantages with this approach. Advantages are simplicity, simple calculations, and fewer parameters in comparison to other methods. Disadvantages are loss of detail and less realism.

2.3 Microscopic Models

Microscopic models attempt to simulate motion of individual vehicles within a system. Typical parameters are position, velocity, and acceleration. Because the behavior of these models is usually governed by a lead vehicle, they are termed "car-following" models or “follower/followee” models [9]. As part of this study, a microscopic simulation model is developed, so more general information about microscopic models is provided here. Figure 4 below illustrates the sequential nature of vehicles in this model where every vehicle $n$ has velocity $= v_n$, length $= l$, position $= x_n$, and distance to next vehicle $= h_n$ for a time $= t$. This method can be used to update the parameters $(v_n, x_n, h_n)$ for all the
vehicles \((n, n+1, n+2)\) for time \(= t + \Delta t\), where \(\Delta t\) is the time-interval used in the microscopic model simulation.

Figure 4: Cars following each other, as explained in [8] (Follower/Followee model)

These models can simulate details of realistic traffic conditions involving human factors. All the vehicles in these models can be classified into two categories. The first category contains the lead vehicle that must adjust to the current state of the traffic light, and the second category contains all of the vehicles following a lead vehicle. Each following vehicle adjusts its acceleration based on the vehicle immediately in front (and also as per the traffic light condition subject to its proximity to the intersection). The adjustment of velocity depends only on the acceleration (or braking power) applied. Similarly, position depends on both acceleration and velocity. This way microscopic models can emulate real world scenarios where a driver can control only accelerator or brake to adjust the speed of the vehicle. Other driver behaviors and parameters such as reaction-time can also be included in these models.
2.3.1 Microscopic Model Assumptions

As microscopic models are complex because of more parameters, there are many assumptions made while running a simulation. Homogenous driver behavior is assumed here: all drivers behave exactly the same way in similar situations. All physical parameters of the vehicles are also assumed to be the same. Ideal traffic conditions (e.g., no accidents) are also assumed. The system can model special safety features to avoid such conditions during a simulation. For simplification, a single lane of traffic is assumed as opposed to a flow of traffic in macroscopic models. Though multiple lanes, turn lanes, etc., can be simulated, our work does not do this.

2.3.2 Microscopic Model Advantages

A primary advantage is the ability to analyze individual vehicles. This helps in modeling a more realistic behavior. For example, driver reaction time can be modeled here to account for the time delay by any human driver in acknowledging an event. Also, macroscopic flow or density can be analyzed with microscopic models.

2.3.3 Microscopic Model Disadvantages

A major disadvantage here is the high computation cost of microscopic models. For discrete-time simulations, as the time interval between steps is reduced, a more accurate simulation of behavior is seen. This, however, results in the simulation running proportionately longer. Perhaps the high computation requirement was the main reason
for the unpopularity of these models in early days of traffic modeling research, and because of that more focus was given to macroscopic models for a long time.

2.4 Mesoscopic Models

Mesoscopic models contain properties of both microscopic and macroscopic models. Mesoscopic models simulate individual vehicles (similar to microscopic models) but describe their activities and interactions aggregately (much like macroscopic models) [11]. These models are more suitable and useful in a traffic network simulation where route based calculations are done. This objection no longer applies for most situations, given the capabilities of modern computers.

2.5 Existing Work and Comparison with Our Method

There are several flow-based adaptive traffic-control systems [5] [6] [7]. These systems help in understanding of the traffic congestion problem and in providing solutions to clear congestion and minimize the time delay while adapting to a better light timings. In [5] the measure of effectiveness is the total control delay which is measured in comparison with the travel time (or time delay) in absence of the control mechanism. In other words, the penalty for the signal is the total transit time if a vehicle didn’t stop at a signal and continued vs. total transit time when it did stop and continue after a delay due to the control signal. Work done in [5] also accounts for the flow density and minimizing overall flow as a measurement.
In a few other types of existing studies [6] [7], macroscopic models use average velocity of vehicles for any optimization. Our method instead focuses on energy usage optimization as the basis for adaptive traffic light control, and energy usage (fuel used per unit time or distance for a vehicle) cannot be calculated properly without considering microscopic parameters of each vehicle. As mentioned in the previous chapter, minimizing fuel usage per vehicle while adapting to a better light timings is the goal of our method. More details of our method are given in Chapter 3.
Chapter 3: Method and Architecture

3.1 Arrival Process

The simulation software used for this study tries to mimic an actual traffic flow. We use a widely accepted Poisson arrival process to simulate realistic traffic.

3.1.1 The Traffic Situation and Different Terms

In the simulation we assume one-way traffic in both North/South (NS), and East/West (EW) directions without any left or right turns. The traffic has a single lane of vehicles in each direction (NS or EW). The farthest visible distance or the horizon from the intersection (traffic signal controller) is the entry point of any vehicle. This also means that any incoming traffic beyond the horizon does not have any impact on the traffic controller or its action. Each vehicle enters the system with maximum allowed “cruising” velocity ($v_c$). Safe distance ($d_{safe}$) between cars can be derived (explained in detail in Section 3.4) as a function of their velocities. As each vehicle enters the system, velocity $v_c$ directly affects the safe distance between the current (entering) vehicle and the vehicle in front (previously entered). An overall picture of all the terms is in Figure 5.
3.1.2 Poisson Arrival Process

The exponentially distributed inter-arrival time is the time between two car arrivals for one direction. The inter-arrival time $t$ depends on the traffic flow density ($\lambda$). There may be a few restrictions on the minimum possible value of $t$, as a very small value will not be safe for the incoming traffic, and a too-high flow of traffic will lead to unbounded queuing hence no steady-state behavior. This parameter, $t$, is the main constituent of the final inter-arrival time between two cars. To provide safe traffic conditions, we bound the inter-arrival time from below to ensure a safe inter-arrival time, i.e., one that guarantees the safety of incoming traffic.

Figure 5: Explanation of different terms in the traffic situation
3.1.3 Delay Parameters

For the incoming traffic, we have an additional parameter apart from time $t$ for the safety aspect. For each incoming vehicle, if the distance to the next vehicle is less than the safe-stopping distance, we add a delay to the entry time of the current vehicle. This delay parameter is the time equivalent to the additional distance required to increase the current distance to a safe distance. Safe stopping distance is explained later in this Chapter in Section 3.4. Derivation of the delay parameter is explained in Figure 6.

For Car_{n+1}
Velocity = v_{n+1}
Safe distance = S_{n+1}
Distance from next = d

\[ t_{delay} = \max(0, \frac{(S_{n+1} - d)}{v_{n+1}}) \]

Figure 6: Explanation of delay parameter
The equation in the Figure 6 calculates the delay in arrival time based on the current position of the previously entered car. If the previous car is at a distance that is more than the safe distance of the current car, then no delay is required; otherwise it is calculated based on the formula given.

The delay parameter (and hence, the safe distance) also includes the reaction-time $R_T$ (also known as response-time). This part of the delay parameter is for the consideration of the human factors in the simulation. A human driver may react after a significant time from an actual event’s occurrence. Reaction-time accounts for the time taken in human judgment and response to the event. Reaction-time allows us to simulate a more realistic traffic pattern comprising possible human traits like errors in judgment or response-delay among others. We hope that this factor will be essentially gone with the likes of adaptive cruise-controlled vehicles, as their reaction-time would be negligible opposed to human reaction-time.

The delay parameter is dependent on the velocity and other parameters, hence it can vary for different vehicles. The reaction-time part of the delay parameter is customizable in the simulation and can be changed but is the same for all vehicles in reported results.

### 3.1.4 Effective Capacity of the Road

By capacity of the road we refer to the maximum number of cars on the road at any given time. In the simulation, a minimum safe distance is always maintained between
any pair of cars (one of them is behind other). We assume that each traffic-light has limited vision, in other words a light’s vision is bounded by a certain distance to its horizon. For simplistic computations, we consider that any car enters the system at a fixed distance (light’s maximum vision distance value) with respect to the light. This is also shown in Figure 5. We use about 1 mile (1600 meters) as the initial distance from the light for every vehicle. The entry time for any vehicle is dependent on many parameters, and subject to variations, but entry-distance is a constant. Entry-distance can also be customized in the simulation software.

The theoretical capacity of the road is derived on the basis of a given $\lambda$ for a period of time. So, if $\lambda$ is 0.5 (0.5 arrivals/sec), minimum safe distance between stopped (immediate) cars is 5 meters (minimum possible safe distance between two cars is assumed to be at least one car length at any given time for any condition), car length is 5 meters, and initial entry point is 1600 meters, then there can be a maximum of 160 cars lined up (and stopped) before the intersection (after a significant time has passed since the starting time). So, the actual maximum capacity of the road comes down to a more realistic and smaller number like 40-45 for the same given parameters. We will talk about this in more detail in the later parts of this chapter and next chapter.

3.1.5 Assumptions in the Arrival Process

In the Poisson arrival process we assume that the rate of arrival doesn’t change over time. So, any disturbance in the flow is not modeled. We also assume that after maintaining the minimum safe distance between arrivals, over all traffic is also safe. In
the simulation, the traffic does not overflow and cars don’t queue up at the entry point. The traffic is ordered, which means no car can jump the queue (no violations are possible because we consider only a single lane of traffic).

3.2 Fuel Loss

For any vehicle, fuel economy is lowest when it is accelerating [15]. Because of the red-light, a vehicle slows down from its cruising velocity, and after the light turns green, it accelerates back to cruising velocity. Though the fuel usage while the vehicle decelerates is low, so it doesn’t contribute much in the overall value, this cannot be viewed as a saving in fuel usage because of the extra fuel required during re-acceleration phase (which is much more than average fuel usage).

3.2.1 Fuel Usage Model

In the fuel usage model that we adopt [14], there are two sets of parameters. The first set comprises parameters for a vehicle pertaining to its physical characteristics such as mass, fuel burn rate, etc. The second set includes traffic related parameters of each vehicle, like position, velocity, and acceleration. For simplicity, the first set of parameters remains the same (and customizable) for each vehicle in the simulation. The equation is as follows:

\[
\Delta F = \begin{cases} 
\alpha + \beta_1 * T_F * v + \left[ \beta_2 * M_v * a^2 * \frac{v}{1000} \right]_{a>0} \times \Delta T & \text{for all } T_F > 0 \\
\alpha \times \Delta T & \text{for all } T_F \leq 0
\end{cases}
\]
\( T_F \) is the total tractive force required to drive the vehicle. \( M_v \) is the mass of vehicle. \( v \) is the instantaneous velocity of any vehicle in consideration. \( a \) is the acceleration of the vehicle. \( \alpha \) is the constant idle fuel consumption rate that applies during all modes of driving. \( \beta_1 \) is the efficiency parameter for the normal fuel usage with a constant velocity. Similarly, \( \beta_2 \) is the efficiency parameter for fuel usage during acceleration phase. \( \Delta F \) gives the fuel usages in \( \Delta T \) time duration for any given car. Over time, \( \Delta F \) is accumulated up to give the total fuel usage of the system, and also normalized fuel usage per vehicle.

### 3.2.2 Assumptions in Fuel Usage Calculation

Among the many assumptions in the fuel usage computation, the first assumption is that all cars have the same physical characteristics. These include mass, tractive force, fuel consumption rate, and fuel consumptions parameters for normal and accelerated motion of vehicles. The mass of the vehicle also includes the mass of the passenger or any other thing inside. Every vehicle has the same maximum allowed velocity and power, so the values of acceleration or braking-power are same across all of them. The braking-power uniformity assumption significantly signifies safe-following-distance calculation [20].

### 3.3 Architecture

The architecture of the simulation software is straightforward. It has Cars, Lights, and Events as different components in the system.
3.3.1 Components

The Car component is instantiated for each car in the system. Each car belongs to either flow of traffic. So, a car can be a part of the North-South or the East-West side of traffic. Each flow of traffic is implemented as an ordered list of Car objects. The car component takes care of the physical and dynamic parameters for all the cars in the system. It has all the physical attributes required for the instantaneous fuel usage calculations and methods to update position/velocity/acceleration or calculate fuel usage.

For one run of the simulation, we have two instances of the traffic Light. Each one is responsible for one direction of the traffic (either North-West or East-West bound traffic). Each Light has a current-state attribute with three possible states: Red, Yellow, and Green. The Light has timings associated with these states as separate attributes. The safety invariant that one of the two Light is Red at all times is maintained by the controller for the simulation.

The Event component is responsible for generating events (entry times for cars) in the system. Every component belongs to the main simulation system, which creates and destroys Car objects. A log of the instantaneous values of Cars’ attributes is also maintained by the main system. Figure 7 shows the setting of the simulation pictorially.
3.3.2 Communication between Components

The main system has Lists of Cars. Each Car can get necessary information from the Car in front of it, and also from the traffic Light. Based on the fetched information, it can then update its acceleration/braking.
3.3.3 Diagram

Figure 7: Diagram depicting the simulation with lists of cars and lights
3.4 Driver Algorithm

For this study, the driver algorithm is the crux of the coding part. Each car in the simulation behaves in physical terms like a real car (velocity can only be changed using accelerator or brake). In the simulation for every time unit (e.g., $\Delta T = 0.1$ second or $\Delta T = 0.5$ seconds), an update function is called for every car. This update function calculates new values for all the attributes of the cars in a fixed order of steps. For a car C let $P$ be the position, $D_{Light}$ be the distance from traffic light, $D_{next}$ be the distance from next car, $v$ be the velocity, $a$ be the acceleration, $b$ be the braking power, $L_c$ be the car length. Let $P_{light}$ be the position of light. Let $R_T$ be the reaction time. To calculate the attributes for car $C_1$, kinematics equations are shown below in Table 1.
<table>
<thead>
<tr>
<th>Equations</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ P_1 = P_1^{old} + v_1 \Delta T + \frac{1}{2} a_1 \Delta T^2 ]</td>
<td>This equation is the basic equation of motion ( s = u \cdot t + \frac{1}{2} \cdot a \cdot t^2 ). Where ( s ) is the distance, ( u ) is initial velocity, ( t ) is time, ( a ) is acceleration.</td>
</tr>
<tr>
<td>[ v_1 = v_1^{old} + a_1 \Delta T ]</td>
<td>This equation is based on ( v = u + a \cdot t ).</td>
</tr>
<tr>
<td>[ D_{\text{Light1}} = P_{\text{Light}} - P_1 ]</td>
<td>This is the distance from the light. So, position of light – position of car.</td>
</tr>
<tr>
<td>[ D_{\text{next1}} = P_2 - P_1 ]</td>
<td>This is the distance from the next car. So, position of car in front – position of this car.</td>
</tr>
<tr>
<td>[ D_{\text{safe}} = L_c + \max \left{ 0, \left( \frac{v_1^2}{2b} - \frac{v_2^2}{2b} \right) \right} + v_1 \cdot R_T ]</td>
<td>This equation derives the safe distance of the car based on its velocity, braking power, reaction-time, car-length, and the velocity of the car in front. This includes the effect due to reaction-time ( R_T ).</td>
</tr>
</tbody>
</table>

Table 1: Equations
Steps to calculate new acceleration are shown in Figures 8, 9, 10, and 11. All the steps mentioned in these diagrams are called for every car, after every $\Delta T$ time units, for the duration of the simulation. Figure 8 contains the main flow of the calculation of acceleration part. It is connected to part A, part B, and part C for few conditions. Part A is shown in Figure 9, part B is shown in Figure 10, and part C is shown in Figure 11.

For each vehicle, we assume different states based on its velocity and acceleration: cruising, stopping (or slowing), speeding, and stopped. Any vehicle reaches cruising state after attaining the max velocity (this also means zero acceleration henceforth). A vehicle with positive value of acceleration is in speeding. A vehicle in stopping or slowing state has a net negative acceleration (i.e. forward acceleration value is zero and brakes are applied). A stopped state would mean that both acceleration and velocity values are zero.

In the main flow (Figure 8), the first condition checks whether the car in consideration has crossed the signal. If it hasn’t, the second condition checks if it is in the light-area (light-area is a customizable distance, from the traffic-light, that tells if the traffic-light is visible from the car’s position). If the car has already crossed the signal, then it goes to part A (Figure 9). If the car is in the light-area, then the third condition checks for the state of the light for RED/YELLOW/GREEN. If the state is RED, then it goes to part B (Figure 10). If the state is YELLOW, then it goes to part C (Figure 11), else it goes to part A (Figure 9).
In part A (Figure 9), the first condition checks if the car is the first car. If it is, then the second condition checks if it has reached its maximum velocity. Based on the velocity value, its state is marked as *cruising* or *speeding*. Similarly if the follower-car’s required-safe-distance ($D_{safe}$) is more than the distance-to-next car ($D_{next}$), then the follower-car’s state is marked as *stopping*. If the follower-car’s $D_{safe}$ is less than its $D_{next}$, then its state is marked as *cruising* or *speeding* depending on its velocity value.
Figure 9: Part A - for GREEN light and for vehicles that have crossed the signal
In part B (Figure 10), the first condition checks if the car is the first car. If it is, then the second condition checks if $D_{safe}$ is less than $D_{light}$. If $D_{safe}$ is less than $D_{light}$, then the state is changed to stopping. The flow marked with * is a guard condition, for the car
that cannot stop with normal braking power, can enable maximum allowed braking power to slam the brakes. This is a special condition and we never expect to see such a case because of the safe-distance and stopping-distance invariants. We considered this special condition because, in the simulation for large values of $\Delta T$ (as opposed to the one we used), it is possible.

If it is not the first car, then based on $D_{\text{safe}}$ and $D_{\text{next}}$ it can decide to cruise or stop.
Part C (Figure 11) is almost identical to Part B except, that the car can speedup, instead of slowing/slamming if it cannot stop before the light. Also for the follower-cars, if the $D_{\text{next}}$ is less than $D_{\text{safe}}$, then they can slam on the brakes instead of just stopping.
After the state of the car is decided, based on that and the previous state, a value of acceleration is decided. This value is in effect for the next $\Delta T$ and is used in the calculation of velocity, and position in the next iteration of the update function.
Chapter 4: Results and Conclusion

4.1 Results

The simulation software is run with different input values for red and green timings. Our idea is to find, for each possible input combination, a light timing that gives minimum fuel usage. We have fixed the yellow light time to 5 seconds for all the runs (this is because, for a vehicle at cruising speed safe stopping time is over 4 seconds) to provide safety. The simulation was run for 1 hour of simulated time for each with different setting, and results after each were compared. After every $\Delta T = 0.5$ seconds, the system updates every parameter for every car in it.

4.1.1 Comparison of fuel usage for different timings

We have run the simulation for different combinations of red light timings for various values of $\lambda$ (same in both directions). In this case, we used total cycle timings of 60 and 50 seconds, and changed red light timing value from 10 seconds to 50 seconds (40 sec in the latter case) in the NS direction (So red light timing in the EW direction changed accordingly). We plot fuel usage for various flow rates for different red light timings. This way we can know if, for the same traffic flow rates and same total cycle time, one red light timing is better than others with respect to fuel usage per car.
Figure 12: Fuel usage for 50 seconds total cycle time with different red-light timings
Figure 13: Fuel usage for 60 seconds total cycle time with different red-light timings

In Figure 12 and 13, not all fuel usages for various flow rates are shown. We observed queue of cars building up when the flow rate was too high in one direction with smaller green time, and in that case, fuel usage doesn’t convey any real interpretation.

On close observation, however, we can conclude that there is no significant change in the fuel usage over various red light settings and no real trend can be generalized here. This conclusion is contrary to our original hypothesis that a saving in
fuel usage could be achieved by changing red light timings for V2I enabled infrastructures. One possible explanation of why we are not able to prove our claim could be the fact that in this study we do not consider the fuel usage for flow rates which build up queue(s) in any or both direction(s). Over time total fuel usage for all queued up cars can significantly change this situation and we might be able to get a meaningful change in fuel usage for various red light timings but that problem will be very hard to model in a simulation software as it involves handling of abnormal (unbounded) flow rates.

4.1.2 Comparison of fuel usage for different cycle timings

Here a comparison of cycle times (for different flow rates) with fuel usage is analyzed. We can find out if one cycle timing is better than others. This comparison is shown in Figure 14.
From Figure 14, we see a clear trend that tells us different total cycle timings (each having 50% red light timing) for similar (equal in our case) flow rates, will not materially affect fuel usage per car.

4.1.3 **Comparison of fuel usage for different flow rates**

In this scenario, we study effect of various red light timings over fuel usage for different flow rates in different directions. We can find out if one red light timing is better than others for a given flow rate ratio ($\lambda_{NS}/\lambda_{EW}$). This comparison is shown in Figure 15.
Total fuel usage per car for different flows ($\lambda_{NS} \neq \lambda_{EW}$) for both NS and EW directions for different red light timings in NS direction (flow rate is in arrivals per sec) for 50 sec total cycle time.

Figure 15: Fuel usage for different flow rates and red light timings for total cycle time of 50 sec.
From Figure 15 we can observe a decrease in fuel usage for few cases. The trend here shows low fuel usage when small red light is applied in the direction with larger flow rate. But, the maximum saving in fuel is not more than 5%.

4.2 Conclusion

From Figures 12 and 13, we see a clear trend that for similar flow rates, various red light timings don’t offer a saving in fuel usage. This is not aligned with our original expectations. On the contrary by these results we can conclude the opposite of our claim; an intelligent light controller (enabled with V2I communications) cannot provide significant savings in fuel (per vehicle) for an isolated intersection, given similar or same flow rates in the two directions.

From Figure 14, we observe a clear and flat trend. Fuel usage values in this part do not vary much with increase in total cycle time and keeping red light timing as 50% of the total. From this scenario, we can conclude various total cycle timing values each with 50% red light timing do not affect fuel usage, given similar (or same) flow rates of vehicles in either direction.

From Figure 15, we can say that if the ratio of flow rates is very small (or large), different red light timings can show a lower fuel usage. This falls in line with our thesis. However, as the savings in fuel usage that we have observed are not much and that alone cannot be used to justify the possible cost of building V2I infrastructures and communications.
In conclusion we can say that, for an isolated intersection having normal, and similar incoming flow rates, an intelligent light controller cannot provide a notable saving in overall fuel usage per vehicle. In the scenario where the ratio of flow rates is very high or low, an intelligent light controller can provide small savings in fuel usage.

4.3 Future Work

In this work, the simulation was limited to a single lane of traffic and no turn was allowed at the signal. In future work, it is possible to include more complex situations with multiple lanes and side turns. Another interesting problem could be to accommodate pedestrians in the simulation. Other than these physical restrictions, there are several areas that can be explored. Some research labs are trying to model Road Frustration Index which comprises stress and frustration of the drivers [19]. It is possible to model the driver-frustration as a metric in the cost function to determine effective traffic light timings.

Many times in real traffic, a long queue of vehicles is formed near an intersection. To avoid this type of problem would be one of the goals of a system like ours, but, modeling it in the system as an anomaly in the traffic flow is a real challenge. Also, modeling the effect of extra fuel usage, when there is a huge queue of cars waiting at a signal for a long time due to congestion, could be equally interesting and challenging. It is also possible to model other types of disturbance due to accidents or problems. Modeling and simulating real world traffic problems can have endless parameters or scenarios.
References


Appendix A: Code
/This class has all the parameters used in the simulation software
 * *************/

public class Constants {

    public static double delay = 0.25; // delay time in sec
    public static int mean = 3;
    /*Assuming a safe distance (in terms of time=3seconds) to provide safety*/
    public static double recTime = 0.5;
    public static int safeFactor = 4;

    public static double initialVelocity = 30;

    //Light parameters
    public static int red = 0;
    public static int green = 1;
    public static int yellow = 2;

    public static int logTime = 5; //Time timings in sec
    public static int cycleTime = 50;

    //x is essentially the time for RED in NS
    public static int x = 20;

    public static double lambdaNS = 0.15;
    public static double lambdaEW = 0.2;

    public static int genYellow = 5;
    public static int yellowTimeNS = genYellow; // yellow light time
    public static int redTimeNS = x;
    public static int greenTimeNS = cycleTime - (genYellow + x);

    public static int yellowTimeEW = genYellow; // yellow light time
    public static int redTimeEW = greenTimeNS + yellowTimeNS;
    public static int greenTimeEW = cycleTime - (yellowTimeEW+redTimeEW);

    public static int mulFactor = 720;
    /*Tried with
     * 720 is an hour
     * 360 is 30 mins
     * 600 is 50 mins
     * 120 is 10 mins
     */

    /*Total running time is maxTime*/
    public static long maxTime = LogTime*mulFactor; //

    public static int fullCapacity = 0;
    public static int fixCapacity = 1;

    /*Fix capacity mode can be used to have different number of vehicles in each side*/

    45
public static int mode = fixCapacity;

public static int NScapacity = 100; /*Maximum number of Cars in NS direction*/
public static int EWcapacity = 100; /*Maximum number of Cars in EW direction*/

public static int LightArea = 200;

//Car specifications
public static double maxVel = 30; //in m/sec
public static int carLength = 5;
public static int maxA = 5;
public static int maxB = -4;
public static int entryPosition = -1600;

public static int LastCarLocation = entryPosition + 200;
public static int mass = 1400;

public static int cruising = 0;
public static int stopping = 1;
public static int speeding = 2;
public static int stopped = 3;
public static int slamming = 4;

//For fuel calculations
public static double alpha = 3.7 * Math.pow(10, -7);
public static double beta1 = 9 * Math.pow(10, -7);
public static double beta2 = 3 * Math.pow(10, -7);

}
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.text.SimpleDateFormat;
import java.text.DateFormat;
import java.util.Calendar;
import java.util.Vector;

public class DataGen {
    private static Vector<Car> listNS = new Vector<Car>();
    private static Vector<Car> listEW = new Vector<Car>();
    private static double fuelNS = 0.0;
    private static double fuelEW = 0.0;
    private static double fuelLost = 0.0;
    public static String fileContent="";
    public static String fileC = "";
    public static String minVC = "";
    public static String fuelData = "";

    /*Lights variables*/
    private static Light lightNS = new Light(Constants.green,"lightNS",
        Constants.redTimeNS, Constants.yellowTimeNS, Constants.greenTimeNS);
    private static Light lightEW = new Light(Constants.red,"lightEW",
        Constants.redTimeEW, Constants.yellowTimeEW, Constants.greenTimeEW);

    private static int numberOfCarsNS = 0;
    private static int numberOfCarsEW = 0;
    private static int totalCars = 0;
    private static double currTime = 0;

    /*Velocity at the time of Entry for each Car. In meters per sec*/
    private static Events e = new Events();

    public static void addNewCar(Vector<Car> cars, double entryTime, Light light){
        Car c = null;
        if(cars.size()==0)
c = new Car(null, light, entryTime, Constants.initialVelocity, 1);
else{
    Car next = cars.lastElement();
    c = new Car(next, light, entryTime, Constants.initialVelocity, 2);
}
cars.add(c);
/*System.out.println("ENTRY TIMES: "+Math.round(entryTime));*/
}

public static double updateCars(Vector<Car> cars, Light lt){
    double f=0.0;
    if(cars.size()!=0){
        for(int i=0;i<cars.size();i++){
            Car cr = cars.elementAt(i);
            cr.update(lt, i, Constants.delay);
            f+= cr.getFuelBurnRate();
        }
        for(int i=0;i<cars.size();i++){
            Car cr = cars.elementAt(i);
            if((cr.getPosition()>0 && cr.getVelocity()==30 &&
            //Object o = cars.elementAt(i).getNext();
            /*if(o!=null && i!=0)
                cars.elementAt(i-1).setNext((Car)o);
            else if(o==null && i!=0)
                cars.elementAt(i-1).setNext(null);*/
            //update next field for previous element
            //remove the element
            fuelLost+= (cars.elementAt(i)).getFuelLost();
            totalCars++;
            minVC=minVC+(cars.elementAt(i)).getMinV()+"\n";
            cars.remove(i);
            //*/
            if(o!=null && i<cars.size())
                cars.elementAt(i).setNext((Car)o);
            else if(o==null && i<cars.size())
                cars.elementAt(i).setNext(null);
            else if(i==cars.size() && cars.size()!=1)
                cars.elementAt(--i).setNext(null);*/
        }
        /*if (i!=0 && cr.getNext()==null)
            System.out.println("NULL");*/
        System.out.println("NULL");*/
    }
    if(cars.size()!=0){
        Car cx = null;
        for(Car c: cars){
            c.setNext(cx);
            cx=c;
        }
    }
    return f;
    /*System.out.println("Inside update cars "+cars.size());*/
}
}
public static boolean maxOut(Vector<Car> list, int capacity, int mode){
    if(list.size()==0)
        return false;
    else{
        if(mode== Constants.fixCapacity){
            if(list.size()>= capacity)
                return true;
            else
                return false;
        }
        else if(mode == Constants.fullCapacity){
            if(list.lastElement().getPosition() < Constants.lastCarLocation)
                return true;
            else
                return false;
        }
        else
            return false;
    }
}

public static void updateLight(double d){
    lightNS.updateLight(d);
    lightEW.updateLight(d);
}

public static double max(double a, double currTime2){
    if(a>=currTime2)
        return a;
    else
        return currTime2;
}

public static double safeTime(Vector<Car> list){
    double s = -1485;
    if(list.size()!=0){
        double x = list.lastElement().getPosition();
        double t = max(0, (s-x)/30);
        return t;
    }
    else
        return 0;
}

public static void runSimulation(){
    double lastEntryNS = 0;
    double lastEntryEW = 0;
    double avgFuel=0;
    double avgFuelNew=0;
    double avgFuelNewest=0;
    //int cntr=0;
    while(currTime<Constants.maxTime){
        fileContent+=e.getNSEvent()+"--- " currTime;
        //System.out.println(e.getNSEvent()+"--- " currTime);
    }
}
if((e.getNSEvent()<=currTime))
{
    double safet= safeTime(listNS);
    double entryTime = max(lastEntryNS,currTime) +
    e.poisson(Constants.lambdaNS) + Constants.recTime+safet;
    e.setNSEvent(entryTime);
    if(!maxOut(listNS, Constants.NScapacity, Constants.mode))
    {
        addNewCar(listNS, entryTime, lightNS);
        numberOfCarsNS++;
        lastEntryNS = entryTime;
    }
}

if(e.getEWEvent()<=currTime)
{
    double safet= safeTime(listEW);
    double entryTime2 = max(lastEntryEW,currTime) +
    e.poisson(Constants.lambdaEW) + Constants.recTime+safet;
    e.setEWEvent(entryTime2);
    if(!maxOut(listEW, Constants.EWcapacity, Constants.mode))
    {
        addNewCar(listEW,entryTime2, lightEW);
        numberOfCarsEW++;
        lastEntryEW = entryTime2;
    }
}

fuelNS += updateCars(listNS,lightNS);
/*System.out.println("----------------"+listNS.size());*/
Cars:"+(listNS.size()+listEW.size())+" Time:"+currTime+"\n"; /*fileContent="----------------"+listNS.size()+"\n";*/
/*System.out.println("----------------"+listNS.size());*/
fuelEW += updateCars(listEW,lightEW);
avgFuel = (fuelNS+fuelEW)/(numberOfCarsNS+numberOfCarsEW);
avgFuelNew = (fuelNS+fuelEW)/(totalCars);
avgFuelNewest = (fuelLost)/(totalCars);
currTime = currTime + Constants.delay;
/*System.out.println("++"+Constants.lightTime);*/
updateLight(Constants.delay);
/*System.out.println("Light":+lightNS.getName()+" currState":+lightNS.getState()+" curr:"+lightNS.getCurrentLightDuration()+" CurrTime":+currTime);*/
/*System.out.println("Light":+lightEW.getName()+" currState":+lightEW.getState()+" curr:"+lightEW.getCurrentLightDuration());*/
if((currTime%Constants.logTime)==0){
    //switchLight();
    //System.out.println("Current Time":+currTime+"n Fuel Lost per car ");
    //System.out.println(" Total cars (Crossed intersection and backed up at cruising speed):"+(totalCars));
    /*DataGen.fileC+= "------- Current Time:"+currTime+"n";*/
    DataGen.fuelData+= "Current Time:"+currTime+" Fuel Lost per car "
    //DataGen.fuelData="Current Time:"+currTime+" Fuel Lost per car "
    //System.out.println(" Total cars (Crossed intersection and backed up at cruising speed):"+(totalCars));
    /*DataGen.fileC+= "------- Current Time:"+currTime+" ++"*/
    DataGen.fuelData="Current Time:"+currTime+" Fuel Lost per car 
    //System.out.println(" Total cars (Crossed intersection and backed up at cruising speed):"+(totalCars));
    if(currTime==150){
        //TODO
        System.out.println(fuelNS);
        System.out.println(fuelEW);
        System.out.println(fuelLost);
        System.out.println(totalCars);
        System.out.println(numberOfCarsEW);
        System.out.println(numberOfCarsNS);
        fuelNS=0;
        fuelEW=0;
        fuelLost=0;
        totalCars = 0;
        numberOfCarsEW=0;
        numberOfCarsNS=0;
        //System.out.println(numberOfCarsEW);
        //System.out.println(numberOfCarsNS);
        }
System.out.println("Current Time:"+currTime+" Fuel Lost per car ------------");
//System.out.println("Over all cars:"+avgFuel+
System.out.println(" #of Cars entered:
"+(numberOfCarsEW+numberOfCarsNS));
//System.out.println("Over passed cars:"+avgFuelNew);
System.out.println("Passed cars:"+avgFuelNewest+" total #of (crossed)
Cars: "+(totalCars));

//extra
public static void printPoisson(){
    for(int i=0;i<100;i++)
    {
        double min=999;
        double max=0;
        double old= e.poisson1(0.5);
        //System.out.print(i+" > ");
        for(int j=0;j<10;j++){
            double x = e.poisson1(old);
            if(x<min)min=x;
            if(x>max)max=x;
            System.out.print(x+" ");
        }
        System.out.println("--"+min+" + "+max);
    }
}

public static void printCheck(){
    /*To see the arrival times*/
    Events e = new Events();
    double max = 0;
    //double min=10;
    double num= 0;
    double safeTime = Constants.safeFactor +
    Constants.recTime; //getRecTime();
    double old = e.poisson1(0.5);
    for(int i=0;i<100;i++){
        num = e.poisson1(old);
        old = num;
        max += (num>safeTime)?num:(num+safeTime);
        System.out.println(max+"  ..  "+num+" - "+(num+safeTime));
    }
}

public static void main(String[] args) throws IOException {
    // for(int i=10;i<45;i+=5){

```java
//Constants.x=i;
DateFormat dateFormat = new SimpleDateFormat("HH.mm.ss_MM.dd.yyyy");
Calendar cal = Calendar.getInstance();
String timestamp = dateFormat.format(cal.getTime());

String info = "Parameters: \n" + "Total Running time:" + Constants.maxTime + "\nRed(NS):" + Constants.redTimeNS + "\nYellow(NS):" + Constants.yellowTimeNS + "\nGreen(NS):" + Constants.greenTimeNS + "\n";
info += "Parameters: \n" + "Total Running time:" + Constants.maxTime + "\nRed(EW):" + Constants.redTimeEW + "\nYellow(EW):" + Constants.yellowTimeEW + "\nGreen(EW):" + Constants.greenTimeEW + "\n";
//info+= "NS capacity:" + Constants.NScapacity + " EW capacity:" + Constants.EWcapacity;

System.out.println(info);
File file = new File("src/logs/CfuelLog_" + timestamp + "_CT_" + Constants.cycleTime + "_NSR_" + Constants.redTimeNS + ".txt");
if(!file.exists()){
    file.createNewFile();
}
FileWriter fw = new FileWriter(file.getAbsolutePath());
BufferedWriter bw = new BufferedWriter(fw);

File filev = new File("src/logs/minV_" + timestamp + "_CT_" + Constants.cycleTime + "_NSR_" + Constants.redTimeNS + ".txt");
if(!filev.exists()){
    filev.createNewFile();
}
FileWriter fwv = new FileWriter(filev.getAbsolutePath());
BufferedWriter bwv = new BufferedWriter(fwv);

//File file2 = new File("src/logs/CLog_" + timestamp + "_CT_" + Constants.cycleTime + "_NSR_" + Constants.redTimeNS + ".txt");
//System.out.println("Started");
/*
if(!file2.exists()){
    file2.createNewFile();
}
FileWriter fw2 = new FileWriter(file2.getAbsolutePath());
BufferedWriter bw2 = new BufferedWriter(fw2);
*/
fileC+=info+"\n";
fuelData+=info+"\n";
runSimulation();

//bw2.write(fileC);
///
bw2.close();

bwv.write(minV);
bwv.close();
bw.write(fuelData);
```
bw.close();
// System.out.println("Finished");

System.out.println("*****************************************************");
currTime=0;
fileC="";
fuelData="";
//
}
}

/**************************************************************************
************ENDS DataGen.java ********************************
**************************************************************************/
/******Events.java class
* This class handles the events that are used in car generation in the system
* **********/

import java.util.Random;

public class Events {
    public double eventNS;
    public double eventEW;
    public long eventLight;

    public Events(){
        this.eventEW=0;
        this.eventLight=0;
        this.eventNS=0;
    }

    public double poisson(double lambda){
        Random d = new Random();
        double x1 = d.nextDouble();
        double lemda = lambda;
        //considering 0.5 vehicles per second or 1 vehicle in 2 seconds.
        return -Math.log(1-x1)/lemda;
    }

    public double poisson1(double old){
        Random d = new Random();
        double x1 = d.nextDouble()*old;
        double lemda = 0.5;
        //considering 0.5 vehicles per second or 1 vehicle in 2 seconds.
        return -Math.log(1-x1)/lemda;
    }

    public int poissonRandom(double mean) {
        int r = 0;
        Random d = new Random();
        double a = d.nextDouble();
        double p = Math.exp(-mean);

        while (a > p) {
            r++;
            a = a - p;
            p = p * mean / r;
        }
        return r;
    }

    public double getNSEvent() {
        return eventNS;
    }

    public void setNSEvent(double entryTime) {
        eventNS = entryTime;
    }
}
public double getEWEvent() {
    return eventEW;
}

public void setEWEvent(double eWEvent) {
    eventEW = eWEvent;
}

public double getEventLight() {
    return eventLight;
}

public void setEventLight(long eventLight) {
    this.eventLight = eventLight;
}
class Light {

    private int state;
    private long lifetime=0;
    private int position=0;
    private String name;

    private int redTime;
    private int yellowTime;
    private int greenTime;

    private double currentLightDuration=0;

    Light(int state,String name, int r, int y, int g){
        this.state = state;
        this.name = name;
        this.redTime = r;
        this.yellowTime = y;
        this.greenTime = g;
    }

    public void updateLight(double d){
        this.updCurrentLightDuration(d);
        if(this.getState() == Constants.red &&
           this.getCurrentLightDuration()==this.redTime)
            this.red2green();
        else if(this.getState() == Constants.green &&
                this.getCurrentLightDuration() == this.greenTime)
            this.green2yellow();
        else if(this.getState() == Constants.yellow &&
                this.getCurrentLightDuration() == this.yellowTime)
            this.yellow2red();
    }

    public void red2green(){
        this.setState(Constants.green);
        this.currentLightDuration=0;
    }

    public void green2yellow(){
        this.setState(Constants.yellow);
        this.currentLightDuration=0;
    }

    public void yellow2red(){
        this.setState(Constants.red);
        this.currentLightDuration=0;
    }
}
public int getState() {
    return state;
}

public int getPosition(){
    return position;
}

public String getName(){
    return name;
}

public void timerReset(){
    this.setLifeTime(0);
}

public void setState(int state) {
    this.state = state;
}

public long getLifeTime() {
    return lifeTime;
}

public void incrementTime(double delay){
    this.lifeTime+=delay;
}

public void setLifeTime(long lifeTime) {
    this.lifeTime = lifeTime;
}

public void changeLight(){
    if(this.getState()==0)
        this.setState(1);
    else
        this.setState(0);
    this.lifeTime=0;
}

public void extendYellowTime() {
    //future work
}

public void extendRedTime(){
    //future work
}

public double getCurrentLightDuration() {
    return currentLightDuration;
}

public void resetCurrentLightDuration() {
    this.currentLightDuration = 0;
}

public void updCurrentLightDuration(double delay) {

58
this.currentLightDuration += delay;
}
}

/***************************************************************************/
/***********************ENDS Light.java ***********************************/
/******************************************************************************/
public class Car {
    //link to next car
    private Car next;
    //link to light
    private Light light;

    private double minV=30;
    private double velocity; // Max velocity is 100 meters per second
    private double acceleration; // Min = -4 and Max = 5 meter per sec2
    //starting value of acceleration is 0
    private double position;
    /*Position is with respect to Center. So a Car at signal has an absolute
    position zero.*/
    private int carLength;
    private double lifeTime = 0.0; //Life time of a Car. Starts as 0.
    private int carStatus; /* not used */
    int carState;
    /* need to think about it. Right now 0 means approaching and 2 means crossed
    // 0 - cruising (normal speed or same speed)
    // 1 - stopping (same as slowing)
    // 2 - speeding */
    //
    String carName="";
    private double fuelLost;

    public double getMinV(){
        return minV;
    }

    public int getCarStatus() {
        return carStatus;
    }

    public void setCarStatus(int carStatus) {
        this.carStatus = carStatus;
    }

    public int getCarState() {
        return carState;
    }
public void setCarState(int carState) {
    this.carState = carState;
}

private double entryTime;

public Car getNext(){
    return this.next;
}

public void setNext(Car c){
    this.next = c;
}

public double max(double a,double b){
    if(a>b) return a;
    else return b;
}

public int max(int a, int b){
    if(a>b) return a;
    else return b;
}

public int min(int a,int b){
    if(a<b) return a;
    else return b;
}

public double min(double a,double b){
    if(a<b) return a;
    else return b;
}

public Car(Car next,Light light, double entryTime2,double velocity,int carStatus) {
    this.next = next;
    this.light = light;
    this.entryTime = entryTime2;
    this.velocity = velocity;
    this.carStatus = carStatus;
    this.acceleration = 0;
    this.position = Constants.entryPosition;
    this.carLength = Constants.carLength;
    this.carName = Double.toString(entryTime2);
    this.fuelLost=0.0;
}

public boolean maxV(double v){
    //return true if reached max velocity
    if(v<Constants.maxVel) return false;
    else return true;
}

public boolean maxA(double a){
    //return true is reached max acceleration
    if(a<Constants.maxA) return false;
    return true;
}
else return true;

public boolean maxB(double b){
    //returns true if reached max braking
    if(b>Constants.maxB) return false;
    else return true;
}

public double effectiveAcc(){
    //
    double ret=0;
    double distanceFromLight=0;
    double distanceFromNext = 0;
    double safeDistanceRequired = 0;
    //int lightState = this.light.getState();
    //storing the previous state
    int prevState = this.getCarState();
    distanceFromLight = this.light.getPosition() - this.getPosition();
    distanceFromNext = this.getNextCarDistance();
    if(this.next!=null)
        safeDistanceRequired = 1.5*this.getCarLength() +
        max(0,(Math.pow(this.getVelocity(),2)/8) - Math.pow(this.next.getVelocity(),2)/8) +
        (this.getVelocity()* Constants.recTime);
    else
        safeDistanceRequired = Math.pow(this.getVelocity(),2)/8;

    if(distanceFromLight < 0){
        /*CROSSED the signal***********/
        if(this.next==null)
            if(!maxV(this.getVelocity()))
                this.setCarState(Constants.speeding);
            else
                this.setCarState(Constants.cruising);
        else
            if(safeDistanceRequired == distanceFromNext)
                this.setCarState(Constants.cruising);
            else if(safeDistanceRequired < distanceFromNext)
                this.setCarState(Constants.speeding);
            else if(safeDistanceRequired > distanceFromNext &&
                    this.next.getCarState()== Constants.speeding)
                this.setCarState(Constants.cruising);
            else if(safeDistanceRequired > distanceFromNext &&
                    this.next.getCarState()== Constants.cruising)
                this.setCarState(Constants.stopping);
        /*assuming after crossing the signal first car won't slow down or stop*/
    }
    else{

if(distanceFromLight > Constants.lightArea){
    //far from light and approaching
    if(this.next==null){
        if(!maxV(this.getVelocity()))
            this.setCarState(Constants.speeding);
        else
            this.setCarState(Constants.cruising);
    }
    else{
        if(safeDistanceRequired < distanceFromNext)
            if(!maxV(this.getVelocity()) )
                this.setCarState(Constants.speeding);
        else if(safeDistanceRequired == distanceFromNext)
            this.setCarState(Constants.cruising);
        else if(safeDistanceRequired > distanceFromNext)
            this.setCarState(Constants.stopping);
    }
}
else{
    //in light zone RED
    if(this.light.getState() == Constants.red){
        if(this.next==null || this.next.getPosition()>0){
            if(this.getVelocity()!=0){
                this.setCarState(Constants.stopping);
                if(safeDistanceRequired > distanceFromLight ){
                    this.setCarState(Constants.slamming);
                    //System.out.println("In RARE CONDITION");
                    //DataGen.fileC+="In RARE CONDITION\n";
                }
            }
            else
                this.setCarState(Constants.stopped);
        }
    }
}
else{ /*for all following cars in RED light zone*/
    if(this.getVelocity()!= 0){
        if(safeDistanceRequired > distanceFromNext ){
            if(safeDistanceRequired < distanceFromLight)
                this.setCarState(Constants.stopping);
            else
                this.setCarState(Constants.slamming);
        }
        else
            this.setCarState(Constants.stopped);
    }
    else{ /*for all following cars in RED light zone*/
        if(this.getVelocity()! = 0){
            if(safeDistanceRequired > distanceFromNext ){
                if(safeDistanceRequired < distanceFromLight)
                    this.setCarState(Constants.stopping);
                else
                    this.setCarState(Constants.slamming);
            }
        }else
            this.setCarState(Constants.slamming);
    }
}
this.setCarState(Constants.cruising);

} //red zone following cars loop ends
} //red zone loop ends

//GREEN ZONE
else if(this.light.getState() == Constants.green){
//Green
if(this.next==null || this.next.getPosition()>0){
  if(this.getVelocity()<Constants.maxVel)
    this.setCarState(Constants.speeding);
  else
    this.setCarState(Constants.cruising);
} else{ /*for all following cars*/
  if(safeDistanceRequired > distanceFromNext)
    this.setCarState(Constants.speeding);
  else if(safeDistanceRequired ==
    distanceFromNext)
    this.setCarState(Constants.cruising);
  else{
    this.setCarState(Constants.cruising);
  }
}
else if(this.light.getState() == Constants.yellow){
//Yellow
if(this.next==null || this.next.getPosition()>0){
  if(safeDistanceRequired > distanceFromLight
    distanceFromNext)
    this.setCarState(Constants.speeding);
  else if(safeDistanceRequired ==
    distanceFromNext)
    this.setCarState(Constants.cruising);
  else{
    this.setCarState(Constants.cruising);
  }
}
else{
  this.setCarState(Constants.speeding);
  //Need to just speedup
  this.light.extendYellowTime();
  //Light can extend its yellow timing
}
else
  this.setCarState(Constants.speeding);

else
  this.setCarState(Constants.stopping);

} else{
  this.setCarState(Constants.stopping);

}
//every other car stops
} //else ends for following cars
} // if loop ends for yellow light
} // else loop ends for light zone

} // else loop ends for traffic before crossing the signal

//Changes in acceleration based on state of the car
switch(this.getCarState()){
    case 0: //System.out.println("Cruising");
        if(prevState == Constants.speeding)
            ret = 0;
        if(prevState == Constants.cruising)
            ret = -1;
        else if(prevState == Constants.stopped)
            ret = 2;
        else if(prevState == Constants.stopping)
            ret = 2;
        break;
    case 1: //System.out.println("Stopping");
        double x=0;
        double y = min(distanceFromLight,distanceFromNext);
        if(y>0)
            x = -(Math.pow(this.getVelocity(),2)/(2*y)+0.2);
        if(x<=-10)
            ret=-10;
        else ret=x;
        break;
    case 2: //System.out.println("Speeding");
        ret = 2;
        if(maxV(this.getVelocity()) && prevState !=
            Constants.stopping)
            ret = 0; //Should not happen though
        else if(!maxV(this.getVelocity()) && prevState ==
            Constants.stopping)
            ret = 1;
        if(safeDistanceRequired < distanceFromNext &&
            this.getVelocity()<Constants.maxVel) ret = 4;
        break;
    case 3: //System.out.println("Stopped");
        ret = 0;
        if(this.getVelocity()!=0)
            ret = -4; //shouldn't happen
        break;
    case 4: //System.out.println("Slamming");
        ret = -10;
        break;
    //case 5: System.out.println("Cruising");break;
}
this.updateFuelLost();
```java
public void changeAcc(){
    setAcceleration(effectiveAcc());
}

public void changeVelocity(double delay){
    double v = this.getVelocity()+(this.getAcceleration()*delay);
    double x=0;
    //System.out.println(delay);
    if(v>=30){x=30;
        setVelocity(x);
    } else if(v<=0){x=0;
        setVelocity(x);
    } else {x=v;
        setVelocity(x);
    }
    if(x<=minV)
        minV=x;
}

public void changePosition(double delay){
    double d = (getVelocity()*delay) + (0.5* getAcceleration() * Math.pow((delay), 2));
    setPosition(getPosition()+d);
}

public void update(Light lt,int index, double delay){
    /*System.out.println("posi:"+this.getPosition()+" v:"+this.getVelocity()+" a:"+this.getAcceleration()+" Entry:"+this.entryTime+" + index:"+index");*/
    //DataGen.fileContent+="posi:"+this.getPosition()+" v:"+this.getVelocity()+" a:"+this.getAcceleration()+" Entry:"+this.entryTime+" + index:"+index+
    changePosition(delay);
    changeVelocity(delay);
    changeAcc();
}

public double getNextCarDistance(){
    return ret;
}
```
if(this.next!=null)
    return (this.next.getPosition() - this.getPosition());
else
    return (this.light.getPosition() - this.getPosition());
}

//not useful
public static Car getPrevCar(Car c, Vector<Car> Q) {
    int index = Q.indexOf(c);
    Car x = null;
    if(index!=0){
        x = Q.elementAt(index-1);
        System.out.println(x.getEntryTime() + " --- "+c.getEntryTime());
    }
    return x;
}

public double getEntryTime(){
    return this.entryTime;
}

public void setVelocity(double velocity){
    //Car x = Simulator.getNext(this);
    //System.out.println(x.entryTime);
    this.velocity = velocity;
}

public double getAcceleration(){
    return this.acceleration;
}

public void setAcceleration(double acceleration){
    this.acceleration = acceleration;
}

public double getPosition(){
    return this.position;
}

public void setPosition(double x){ this.position = x;}

public double getFuelBurnRate() {
    //Returns the Fuel burnt per unit time.
    double f;
    if(this.getAcceleration()>0) // && this.getVelocity()<30)
        f = (Constants.alpha + (Constants.beta1* this.getVelocity()) +
            Constants.beta2*Constants.mass*Math.pow(this.getAcceleration(), 2)*this.getVelocity())*
                Constants.delay;
    else f = (Constants.alpha + (Constants.beta1* this.getVelocity()))*Constants.delay;
    return f;
}

public double getVelocity() {
    return this.velocity;
}


```java
public int getCarLength() { return carLength; }

public double getLifeTime() {
    return lifeTime;
}

public void setLifeTime(double lifeTime) {
    this.lifeTime = lifeTime;
}

public double getFuelLost() {
    return fuelLost;
}

public void setFuelLost(double x) {
    this.fuelLost = x;
}

public void updateFuelLost() {
    double x = this.getFuelLost() + this.getFuelBurnRate();
    //x = x + this.getFuelBurnRate();
    this.setFuelLost(x);
}

/****************************ENDS Car.java *********************************************/
Appendix B: Sample of the log file
Parameters:
Total Running time: 3600
Red(NS): 20 Yellow(NS): 5 Green(NS): 25
Parameters:
Total Running time: 3600
Red(EW): 30 Yellow(EW): 5 Green(EW): 15

Current Time: 5.0 Fuel Lost per car --------------
Over all cars: 0.2625007634375 total #ofCars: 4
Passed cars: NaN total #ofCars: 0
Current Time: 10.0 Fuel Lost per car --------------
Over all cars: 0.466699245833333 total #ofCars: 6
Passed cars: NaN total #ofCars: 0
Current Time: 15.0 Fuel Lost per car --------------
Over all cars: 0.7916764089583334 total #ofCars: 6
Passed cars: NaN total #ofCars: 0
Current Time: 20.0 Fuel Lost per car --------------
Over all cars: 0.8388967954166671 total #ofCars: 9
Passed cars: NaN total #ofCars: 0
Current Time: 25.0 Fuel Lost per car --------------
Over all cars: 1.0600106747499995 total #ofCars: 10
Passed cars: NaN total #ofCars: 0
Current Time: 30.0 Fuel Lost per car --------------
Over all cars: 1.3950131478749999 total #ofCars: 10
Passed cars: NaN total #ofCars: 0
Current Time: 35.0 Fuel Lost per car --------------
Over all cars: 1.5909228411363634 total #ofCars: 11
Passed cars: NaN total #ofCars: 0
Current Time: 40.0 Fuel Lost per car --------------
Over all cars: 1.5643009464285715 total #ofCars: 14
Passed cars: NaN total #ofCars: 0
Current Time: 45.0 Fuel Lost per car --------------
Over all cars: 1.6812662734375 total #ofCars: 16
Passed cars: NaN total #ofCars: 0
Current Time: 50.0 Fuel Lost per car --------------
Over all cars: 1.7707354182025623 total #ofCars: 18
Passed cars: NaN total #ofCars: 0
Current Time: 55.0 Fuel Lost per car --------------
Over all cars: 2.069283645221551 total #ofCars: 18
Passed cars: 3.4403725514680428 total #ofCars: 1
Current Time: 60.0 Fuel Lost per car --------------
Over all cars: 2.3038526490278843 total #ofCars: 18
Passed cars: 3.5049314661627924 total #ofCars: 3