For office use only
T1 ________________
T2 ________________
T3 ________________
T4 ________________

For office use only
F1 ________________
F2 ________________
F3 ________________
F4 ________________

2015

18th Annual High School Mathematical Contest in Modeling (HiMCM)
Summary Sheet

(Please attach a copy of this page to your Solution Paper.)

Team Control Number: 6117

Problem Chosen: A

Lane closures are one of the most common sources of traffic congestion, which can lead to dramatically increased driver anger and slower commute times for everyone. In order to improve this situation, our team analyzed the possible outcomes of a lane closure scenario and determined optimal strategies, both on the individual and systemic level, to improve traffic flow and prevent road rage. We applied Monte Carlo and fluid dynamic methodologies to create a robust simulation and experimented with countless different scenarios. Through our model, we have determined a series of guidelines that one individual can follow to improve his commute time and overall traffic flow, as well as guidelines, that if implemented on a larger scale, could result in much less traffic congestion for everyone.

When creating our model, we sought to make it realistic and able to test the multitude of scenarios that one may encounter, and determine best practices for each. Our model was primarily a Java simulation, comprising the given road with a lane closure, and populated with car objects. As in real life, different cars may have different driving styles, but our test driver was programmed to follow specific strategies and determine whether they were effective. Since the problem gave a very generalized situation, we found it more tractable to work with additional specificity, testing all possible scenarios that composed the general case. In this manner, our model became very robust and could test situations or strategies that were unexpected or unintuitive. We applied Monte Carlo principles of testing all possible scenarios to determine the optimal strategy.

We also looked toward fluid dynamic principles for insight. The Darcy-Weisbach equation, which models the outflow of laminar fluid towards an opening, was modified for our purposes to accurately model the analogous outflow of non-laminar cars towards an opening -- the unblocked lane. Instead of finding the pressure of the fluid at various points in the tube, we could calculate congestion, the parallel to pressure for our modified equation, at those points, and our test driver would act accordingly to reduce congestion in the system. In this way, we determined not only generalizable guidelines for optimal actions in any scenario, but also scenarios or strategies that would reduce congestion as a whole. As anger is based on congestion

and long commute times, reducing congestion also helped us minimize anger and prevent road rage.

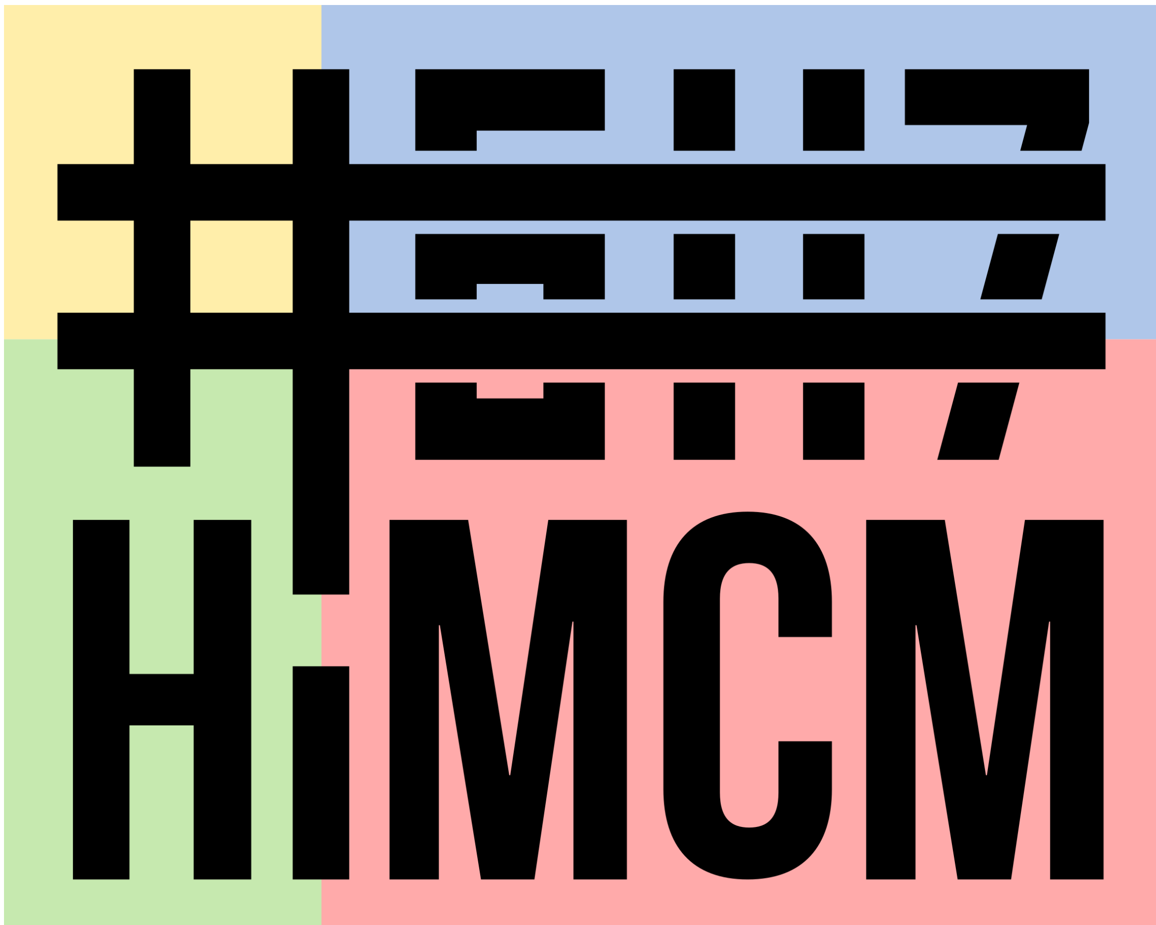
We found that a congestion-balancing strategy, where an individual driver is conscious of changes in congestion that result from his possible actions and acts accordingly to balance , was most effective in reducing both an individual's transit time as well as the anger of all drivers. Were a given driver to employ this strategy, we found that his transit time decreased by 8% and overall driver anger decreased by 10%. By moving to less congested areas, a given driver not only was able to speed up and take advantage of that space, but also freed up space for drivers behind him. This strategy proved most effective both on two-lane and three-lane scenarios, with both 35 and 65 mph speed limits.

Based on our findings, we developed guidelines to include in the DMV driver education materials, so that our conclusions, through education and outreach, can be implemented on a larger scale to improve traffic flow. We have faith that, through following our recommendations, traffic jams due to lane closures may be greatly alleviated.

Go With The Flow

Team 6117

November 2015



Contents

1 Introduction **4**

2 Assumptions **6**

3 Model **9**
3.1 Goals of Model . . . 9
3.2 Summary of Program . . . 9
3.3 Influences . . . 9
3.3.1 Monte Carlo Methods . . . 9
3.3.2 Water Flow . . . 10
3.4 Analysis of Variables . . . 14
3.5 Model Building 1.0 . . . 16
3.5.1 Considerations . . . 16
3.5.2 Congestion and Anger . . . 17
3.5.3 Congestion Balancing Strategy . . . 18
3.5.4 3 Lanes to 2 Lanes . . . 20
3.5.5 3 Lanes to 1 Lane . . . 20

4 Model Results **21**
4.1 Test Driver (Sedan) . . . 21
4.2 Speed Limits . . . 22
4.3 Sign Distance . . . 22
4.4 Test Truck . . . 23
4.5 Multiple Test Drivers . . . 24
4.6 3 Lanes to 2 Lanes . . . 24
4.7 3 Lanes to 1 Lane . . . 25

5 Implications of Model **26**
5.1 Congestion Balancing Strategy . . . 26
5.2 Efficiency vs. Fairness . . . 27
5.3 Trucks on Highways . . . 28
5.4 Sign Distance . . . 28
5.5 Game Theory Implications . . . 29
5.6 Guidelines and Signage . . . 29

6 Strengths and Weaknesses **30**
6.1 Strengths . . . 30
6.2 Weaknesses . . . 31

7 Conclusion **32**

1 Introduction

In a famous incident in August 2010, a Chinese traffic jam stretched for over 100 kilometers and stranded drivers took over two weeks to get from one side to the other. Stemming from a routine lane closure for road maintenance, the traffic jam soon spiraled out of control, as cars piling up at the jam greatly outnumbered those leaving the other end. Sadly, in todays overpopulated world, these sorts of nightmare scenarios are only becoming more and more common, from chronic traffic jams in congested Chinese cities to Carmageddon in Los Angeles. For drivers, these extensive traffic jams can be frustrating at best and downright infuriating at worst, leading to potentially unsafe driving and the chance of road rage (1).

Traffic jams are practically an epidemic in the modern world. As with any sub-optimal system, they produce countless undesirable consequences. By driving at a slower pace, with repeated accelerations and decelerations, each car must spend much more fuel to travel the same distance — a drawback to both the environmentally and economically conscious. Socially, extended commute times and countless hours stuck in traffic lead to not only decreases in an individuals well-being and quality of life, but also greatly reduced productivity. These consequences make it imperative that measures be taken to optimize the system and alleviate these negative effects.

In this paper, we will analyze the behavior of one of the biggest causes of traffic blockages — lane closures. Whether construction causes the blockage of a highway lane or two lanes merge into one on a rural road, the loss of a lane results in congestion and reduced travel efficiency. Upon seeing the traffic sign warning them of an upcoming closure, drivers must immediately adapt their behavior, either changing into the unblocked lane or being mindful of other cars wanting to switch into their lane. If they are not careful, a lane change could easily result in an accident. Drivers may approach the lane merge in several different manners. For example, conscientious drivers may try to merge to the open lane as soon as possible, making sure to leave plenty of space and allow other cars in. Aggressive drivers, on the other hand, may switch lanes whenever they see open space, driving faster but potentially cutting others off in the process.

We will determine the implications of the interactions between various driving styles and how they affect traffic flow as a whole. We wish to find an optimal strategy or driving style that makes driving in this scenario both fair and efficient. On the individual level, we can determine what strategies we can personally employ to improve our own efficiency and traffic flow overall. In the real world, we cannot control the behavior of other cars on the road, but rather only our own actions. Since we want to tackle the general case of a lane closure, our model must be robust and able to simulate the situation based on varying parameters of initial conditions. Based on our findings, we can then decide on an official guideline for driving in this situation that

can be incorporated in the Department of Motor Vehicles driver education materials. In this way, our work will be the first step towards ensuring not only our personal improvement of traffic flow and speeds, but also ensuring optimal travel times for everyone. If our guidelines become widely implemented, hopefully, traffic jams will soon become a thing of the past.

2 Assumptions

Assumption 1: All drivers see the first sign and acknowledge its presence.

Justification: The Federal Highway Administrations (FHWA) Manual on Uniform Traffic Control Devices follows the Sign Legibility guidelines, and all traffic signs are constructed based on these guidelines. Rigorous research and experimentation has been done to determine the optimal placement of signs so that they are legible and guaranteed visible by drivers. The guidelines ensure that the sign will be in the line of sight, contrasting, and noticeable. However, we cannot assume that drivers will choose to act on the sign after it is seen.

Assumption 2: All drivers understand the rules of the road and the meaning of signs.

Justification: In order to drive legally, one must either be a licensed driver or a learners permit holder. In both cases, one must have first passed the rules of the road test. Therefore, they acceptably understand both the rules of the road and meaning of signs.

Assumption 3: Drivers will not necessarily follow given speed limits.

Justification: From a typical highway driving experience, it is clear that many people do not follow posted speed limits. A 2001 survey by the National Highway Traffic Safety Administration found that only 30% of drivers identify themselves as non-speeders and 70% of drivers identify themselves as sometimes-speeders or always-speeders. Clearly, not everyone follows the speed limits at any given time.

Assumption 4: Drivers have different, categorizable driving styles.

Justification: As the speed limit statistics clearly show, not all drivers drive in the same way. Aside from solely speed relative to the speed limit, the driver population will have variations with regard to other attributes as well. The typical driver demographic can be categorized into specific, dichotomous styles (e.g. fast versus slow, selfish versus fair). We will assume that these categories are dichotomous for drivers on the road, so a driver is either slow, leaving ample space between him and the next driver, or fast, speeding up as much as possible. Research has determined that fixed driving styles, based on specific strategies and behaviors, do exist,and in fact, they

can be inherited from parent to child.

Assumption 5: Drivers are able to determine the driving styles of nearby cars.

Justification: Since the drivers have passed their driving test, are licensed, and have not yet crashed, they should be reasonably aware and observant of what is going on around them. For a continuous road about to merge, we know that the drivers on this road have been driving with those around them for a sufficient enough time to be able to observe and understand the driving styles of those around them.

Assumption 6: The blockage is on the right hand side of the road.

Justification: The Problem Statement depicts a situation where the right lane is closed. Since drivers drive on the right side of the road in America, construction will typically close the right side of the road, so construction workers can use the greater space to the right of the road as opposed to the space to the left of the road that is typically occupied by another road flowing in the opposite direction. If a scenario calls for the left lane to be closed (e.g. left lane construction, road block, countries where one drives on the left-hand side of the road), our assumption is not limiting and can be applied to this situation as it is symmetrical. We only work with a optimal road and a suboptimal, blocked road. The orientation doesnt matter.

Assumption 7: All vehicles on the road fall into one of four categories. Vehicles in the same category have similar specifications and can be accurately modeled with generalized specifications for that category.

Justification: In order to make the problem more tractable, we do not need the irrelevant complexity of different makes and models on the road. Although there is variation in specifications between different cars in the same category, the system of classification ensures that cars in the same category have small variation in size, acceleration, and handling, compared to the other categories. The Federal Highway Administration (FHWA) vehicle classifications will adapted into the four major types of vehicles on the road: motorcycles, passenger cars, vans, and trucks. Based on this classification, cars in the similar category will drive similarly. Generalized data is calculated from taking the average of the five most popular cars in each category.

Assumption 8: Drivers do not want to crash.

Justification: People inherently value their lives and well-being. They also value their property (i.e. their cars), and even under extreme road rage, are rational enough to try to avoid risking their lives and their property.

3 Model

3.1 Goals of Model

We want our model to help us in addressing the following concerns:

- Decide the optimal strategy of a particular driver and how to optimally interact with a variety of circumstances and nearby driving styles.
- Simulate repeatedly the lane closure scenario under all likely possibilities to find the best approach on both the individual and systemic level.
- Determine the overall guidelines that drivers should follow when approaching a lane closure to ensure fair, efficient driving.
- Establish a distinction between fair and efficient driving scenarios.

3.2 Summary of Program

We wrote a program in Java that modeled the travel of cars along a road with a lane closure. The program was a detailed simulation that took into account many factors, including distribution of types of cars and drivers, length of road, etc. The simulation could be changed based on all of these factors, as well as the number of cars on the road, number of lanes, and speed limit, all of which allowed complete generalization of our lane closure scenario. Interactions between cars, including our test driver, determined the speed at which cars travelled and the time necessary to bypass the closure. The simulation would continue until the set number of cars we had created had all gone through. Based on this simulation of a lane closure, we can test various factors and personal strategies and determine the time it takes to get through the stretch of road. The full code of our simulation, written in Java, can be found in the Appendix.

3.3 Influences

3.3.1 Monte Carlo Methods

In statistics and many other fields, Monte Carlo methods are commonly used, taking in inputs of random variables in order to receive a better understanding of the overall system. They are a useful tool for optimization, as computing power allows for the testing of large quantities of different situations based on a random distribution.

In our case, we sought to deal with the lack of specificity and lack of information that we encountered with regard to many important variables, including number of

cars, distance from sign, etc. We tackled this by employing Monte Carlo principles – instead of assuming specific information, we could try all possible information and try to generalize overall principles. We could run our simulation countless times, each time inputting different values for the parameters of our simulation, and observe the overall result both for each scenario and the generalizable whole. Based on our variables, we could create a function of the simulation, f(x1, x2 xi) and vary each xi between a sensible range. This would allow us the specificity we need to run individual scenarios while making sure every possible scenario is covered and it is generalizable. Because we want to optimize the lane merger process, we must analyze it with respect to all the different possible scenarios that could potentially occur. Note that sensitivity analysis is inherently incorporated as a result of the Monte Carlo method, as we make small variations in our initial parameters and study the results.

3.3.2 Water Flow

In fluid dynamics, the Darcy-Weisbach equation models water flowing out of a pipe. The model utilizes the assumption that water flow is laminar, and finds an equation based on the pressure differential. We can apply this model to our lane closure scenario: cars in two lanes merging and exiting as one should behave similarly to water exiting through a pipe. However, we cannot use the existing equation because of its assumption that fluid flow is laminar. The flow of cars is clearly not laminar, and thus, our equation must be adjusted accordingly.

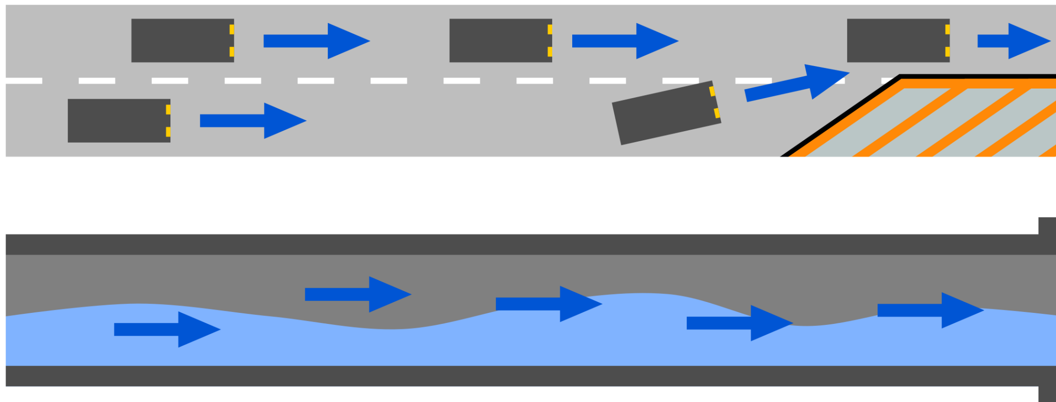


Figure 1: Illustration depicting the analogous situations of water flow and traffic flow.

For water in a pipe, the Darcy-Weisbach equation states that $P = \frac{fv^2\rho L}{2D}$, where P

is the pressure differential at the two ends of the pipe, f is a friction factor, v is the velocity at which the water leaves the pipe, ρ is the density of water, L is the length of the pipe, and D is the diameter of the pipe. To adapt this model to our analogous situation of traffic, we must redefine our variables to be relevant to our scenario and modify the equation accordingly.

Instead of pressure in a pipe, P can be modified to C, the average congestion on given section of road. We must make this adjustment, because the flow of cars is not laminar. Thus, using a pressure differential at the two endpoints does not account for the behavior of the flow in between these two points. C will still yield significant values, as the average congestion throughout a section is a good gauge of how quickly traffic can move. We can disregard f, since it is unique to fluid flow. V will become the average velocity of cars in a given section rather than water flow. ρ will become the average density (cars / area) of cars in the section of interest. L is the length of the section of interest, and D is the width of the section of interest.

Next, we must verify that using these new meanings for variables result in the desired relationships between variables. That is, increasing or decreasing a variable on the right side of the equation, the independent variables, should have the correct effect on the variable on the left side of the equation, the dependent variable. If not, we must make adjustments according to ensure the correct direct or inverse relationships between variables. The following table indicates the desired relationships, relationship given the current equation, and the modifications necessary.

Variable	Does the current equation result in an increase or decrease in congestion (C)?	Should congestion (C) increase or decrease if variable is increased?	Rationale	Necessary adjustment made to correct error
v	increase	increase	If the average velocity of cars in a given section of a road is relatively high in comparison to the rest of the sections on the road, these cars will have to adjust their speed greatly, indicating this section will be more congested.	n/a
ρ	increase	increase	If there are more cars per unit of length on the road, cars have less space to move. Thus, the road will be more congested.	n/a
l	increase	decrease	If you increase the length of the road given a constant density, cars will have more space to move, resulting in less congestion.	Move l to denominator
d	decrease	decrease	Increasing the diameter (increasing the number of lanes) at which cars can leave will create more space for cars to be able to leave, thus clearing congestion on roads and keeping traffic moving smoothly.	n/a

These modifications are necessary as a result of a redefinition of variables. For example, the P involved taking a difference. C involves taking an average, which involves taking a sum. Thus, it is no surprise that certain variables have the opposite effect on C. The underlying behavior should remain the same, however, just adjusted based on our new definitions. Our modified Darcy-Weisbach equation is as follows:

$$C = \frac{\rho v^2}{2ld}$$

We can implement this formula into the logic of the test driver to influence the test driver to make decisions that will decrease C. Specifically, the test driver will look at 5 congestion measurements: the section in front of him in the same lane, the section behind him in the same lane, the section in front of him in the other lane, the section behind him in the other lane, and the section from the car furthest from the merge to the merge itself. The driver will then compare the congestion at these different sections to the overall congestion of the road. The driver will make decisions and act in order to decrease the overall congestion throughout the road. This will ultimately better the flow of traffic, which then will decrease road rage.

Road rage is heavily influenced by environmental conditions, which in this case are the traffic conditions. People become frustrated when their lane is not moving even though the other lane is. That is, people get angry when they are in a situation that is worse off than those around them. With this in mind, we can develop a formula to gauge a drivers anger level. We already have the formula

$$C = \frac{\rho v^2}{2ld}$$

that describes traffic conditions, specifically the congestion on a given section of the road. Thus, we can relate congestion to anger. We can calculate the congestion of the section a given driver is in, say C_s, and compare it to the congestion of the rest of the road, say C_t. A difference of these two values, specifically $C_s - C_t$, will tell us whether a given driver is in a better or worse relative situation. If the driver is in a better situation, he will naturally feel less angry. If the driver is in a worse situation than average, he will feel frustrated at his situation and naturally feel more angry. By taking the integral of $C_s - C_t$ from 0 to τ, when the lane merge process ends, we can calculate the total change in anger level after the entire lane merge process. This integral by t then results in the average change of anger level throughout the entire process. Thus, the formula

$$\frac{\triangle A}{\triangle t} = \frac{1}{\tau} \int_0^\tau C_s - C_t dt$$

can be used in our model to evaluate how going through the lane closure situation will affect a drivers anger level, and thus, whether a driver will experience road rage.

In addition, the overall anger by all the drivers that pass through this stretch of road can serve as a good measure of how frustrating and congested the road is.

3.4 Analysis of Variables

Type of Car: This refers to the classification of a given car based on our assumption that all cars on the road can be classified into four types: motorcycles, passenger cars, vans, and trucks. The type of car determines its max speed, as well as rates of acceleration and deceleration. For example, a truck will take much longer to brake to a stop than a passenger car. The type of car will affect secondary characteristics of the vehicles, of which most important is length of vehicle. Based on our assumptions, the length of the four types of vehicles are listed as followed, calculated by taking the average of the 5 most popular vehicles of each category on the road and rounding to the nearest meter.
Motorcycles: 2 meters
Passenger Cars (Sedans): 4 meters
Vans: 6 meters
Trucks: 20 meters

Type of Driver:
Through research, we have determined six major types of drivers, who have natural tendencies to drive in particular styles. These styles are dichotomous but not exclusive. For example, a fast driver may also become aggressive if anger reaches a certain threshold. Rather, these types are simply guidelines, strategies that given drivers will employ by default, when created. However, they may change based on the surroundings. The types of drivers are listed as follows, with their unique characteristics:

- Fast: A fast driver will accelerate whenever there is any open space in front of him. If he starts in the left lane, he will stay in this lane. If he start in the right lane, he will merge as late as possible. He is more concerned with speed over lane switching.

- Slow: A slow driver will decelerate to leave a minimum space of 7 meters between himself and the car in front of him. He will merge as late as possible and is also more concerned with speed over lane switching.

- Aggressive: An aggressive drivers focus is to keep advancing forward. If there is space in front of him, he will accelerate. If there is space in the lane next to him, he will switch lanes. However, because he is aggressive, his estimate of the space necessary to switch lanes is less than the actual space necessary to switch.

Thus, he is more prone to cutting other drivers off and crashing. Aggressive drivers are more concerned with lane switching over speed.

- Safe: A safe drivers focus is minimizing risk. Thus, he will stay in his lane whenever possible. If he does need to make a switch, he will only do so when there is a lot more space than is actually necessary to make the switch in order to prevent causing accidents. As a safe driver, he likes to go with the flow and will try to match the speed of traffic.
- Fair: Fair drivers act like safe drivers. However, they are more interested in the cars around them when he first sees the sign. If drivers that started behind him end up in front of him, he will not let them merge. If a car that was initially in front of him wants to merge, he will will let him merge. In order to preserve order, he will travel at the same speed as the other traffic in the other lane if he is in the right lane. Selfish: Selfish drivers want to minimize their own personal travel time and be the first one out, even if it increases traffic congestion as a whole. Thus, selfish drivers will try to beat out other cars.

Number of Cars on Road: The number of cars on the road is a variable in the scope of our simulation, representing the total number of cars in the stretch from the beginning of the sign to the lane closure. This variable can be changed to represent different scenarios from a sparsely-trafficked road up to a complete traffic jam. The more cars there are on the road, the slower the speed of each car must be to ensure a safe following distance and prevent collisions.

Distance of Sign from Closure (meters): While the Problem Statement gives examples of the distance between the sign and the lane closure, it does not specifically mention when the first sign appears. This data is crucial, as the further away from the closure that drivers are notified, the more likely that they are able to safely and efficiently switch into the proper lane before the closure. If they are not given enough time and distance to switch, there may end up being a pileup in front of the blocked lane as drivers stuck in that lane try to switch, an unsafe and inefficient situation.

Anger Index: During the lane closure scenario, a given driver may become angrier as a result of his surroundings and events that he encounters. For example, if a lane is heavily congested and not moving, a busy stockbroker may become extremely angry that he will miss his meeting. The anger index is a dimensionless unit that serves as a relative indicator of how angry a person is, where 0 is perfectly neutral. The higher the anger index is, the more angry a person is.

3.5 Model Building 1.0

3.5.1 Considerations

We decided to create a mathematical model that could represent the scenario in order to be able to perform quantitative analysis and more definitively analyze the effects of different driving strategies on the overall anger index and the average time taken by each vehicle throughout the merge process. Our model was designed to take into account the many complex factors that make up any lane merging situation and then simulate the actions of every driver on the road in responses to those environmental stimulus.

Rather than base the model in simply mathematical probability or an analysis of the variables, we decided to replicate the logic of each driver on the road in order to account for differences in driving styles. Simple artificial intelligence algorithms were developed to represent the different types of drivers, each of which represent a substantial portion of the population and altogether make up most of the kinds of drivers on the road at any time, based on our research. Then, we created a road upon which the cars, including our test car, would drive upon. This road was composed of multiple arrays, one for each lane of the road, of length 3000. Every space in an array corresponded to one meter of space, and cars of different types, Sedans, Vans, or Trucks, were made to take up different amounts of space in each array.

For each of these types of vehicles, there was a driving style associated with it as well which would contain the logic used to make each decision made while navigating the merging situation. The road would be updated every second and each of the vehicles would be adjusted and allowed to perform any of the following driving maneuvers: hold, speed up, slow down, merge. Over the course of the simulation, each vehicle on the road was updated and shifts down the road closer to the merge itself.

After all cars have exited the merge and have traversed the full three kilometer stretch of road, the simulation calculates a series of summary statistics for that particular run. These include an average anger index of all the cars, which implies generally how much angrier any particular driver became while going through the merge in that specific case, a total time for the simulation, which helps us gauge the overall efficiency of merge in that specific case, and the average time taken by all of the drivers on the road, which helps us gauge how any individual was affected by the merge.

Using this kind of model, we are allowed a great degree of flexibility when it comes to analyzing different situations, since we could easily modify the number of vehicles on the road, style of drivers present, placement of the roadblock, along with a number of other factors. The model ran any given situation numerous times in

order to determine a general set of output values that could be associated with those specific input factors. The model could then be used to determine what combination of factors resulted in the optimal flow of traffic through the merge, which allowed us to develop a strategy to cause those factors to appear given any starting set of conditions in the model. In a sense, this model was designed to show us what effect any particular factor had on the overall result, which we used to create a strategy which we eventually implemented into the model itself in order to determine if it worked as expected.

3.5.2 Congestion and Anger

A very important aspect of our model was the presence of congestion in the scenario and the effect that it had on the overall anger of the drivers in it. To be able to analyze this, we needed to have a quantitative measure of the congestion in any particular segment of the road. Based off of a model previously developed for determining the pressure of water being forced through a pipe, we developed the following formula for congestion, discussed earlier:

$$C = \frac{v^2 \rho}{2Ld}$$

This formula gives a value for the congestion in a part of the road based off of the number of vehicles present (p), the average velocity of all vehicles in the section (v), the length of the section (l), and the number of lanes over which the section is placed (d). The model calculated a congestion value at every second during the simulation, and uses this to determine a number of other statistics, and is used by our Congestion Balancing Strategy to find the optimal maneuver to make in order to relieve the overall congestion in the road.

We also found that congestion had a direct link to anger felt by drivers, which led us to define an anger index, the degree to which a driver is becoming more angry after each consecutive second spent in the merge scenario. Drivers become angry not necessarily when they need to pass through a congested section of road, but rather when they expect to pass through a less congested section and are forced to wait in one that is more congested. This is defined as the difference between expectations and reality of the driver. Analyzing this anger for a driver over the total time spent in the simulation yields:

$$\frac{\triangle A}{\triangle t} = \frac{1}{\tau} \int_0^{\tau} C_s - C_t dt$$

Where Ct refers to the overall congestion, or the expectation that the driver has, and Cs refers to the congestion in the 100 meter section that the driver is currently

in. The model computes an anger index for each driver across the total length of the simulation and then calculates an overall average anger index, which refers to how much angrier drivers became when traversing the merge under those specific conditions.

3.5.3 Congestion Balancing Strategy

Using the congestion values and the anger index yielded by the simulation, we were able to determine the effects of various factors on the drivers present, which we then used to come up with a general strategy for reducing anger throughout the entire road which could be employed by a single person. Basically, we determined that drivers become more angry when the congestion that they experience is significantly different from what is being experienced throughout the road. This contributes to the anger index, which is an indicator of road rage. Thus, in order to reduce anger, a driver need only do whatever is possible to balance the levels of congestion throughout the road.

This method of balancing the congestion accomplishes multiple goals. The first is an anger reduction, which is a direct result of large differences between congestion in different parts of the road. However, balancing the congestion in the road also allowed drivers to escape the merge quicker, because they were usually moving to areas of lower congestion from areas of higher congestion, which would mean that they were able to speed up and more more quickly out of the scenario.

This concept opens up considerations of a number of maneuvers that could be taken by the test driver in order to both increase his personal efficiency and reduce anger levels throughout the road. In a case where there is a high level of congestion in the right lane, this strategy would yield that one should merge left at the earliest opportunity, which makes logical sense. However, this strategy also implies that merging right if the congestion in the left lane is high, even though the right lane contains a roadblock, is the best option, which goes against what many people might expect. We postulate that while the test driver is a sufficient distance away from the roadblock, merging right is actually the optimal maneuver since it allows one to speed up and reduces the imbalance of congestion throughout the road.

The test car, otherwise the car that is trying to reduce road rage and speed up the merging process for everyone, employs our Congestion Balancing Strategy in order to achieve these goals. In order to make this model applicable to real life, we needed to come up with a mathematical function that could conceivably be put to use in real life. Therefore, it needed to be based upon values that could be determined by a human in a split second by simply observing the environment around their car. Then, the logic used to decide which maneuver to complete also had to be simple enough to be used by a human in a split second in order to make quick decisions while driving.

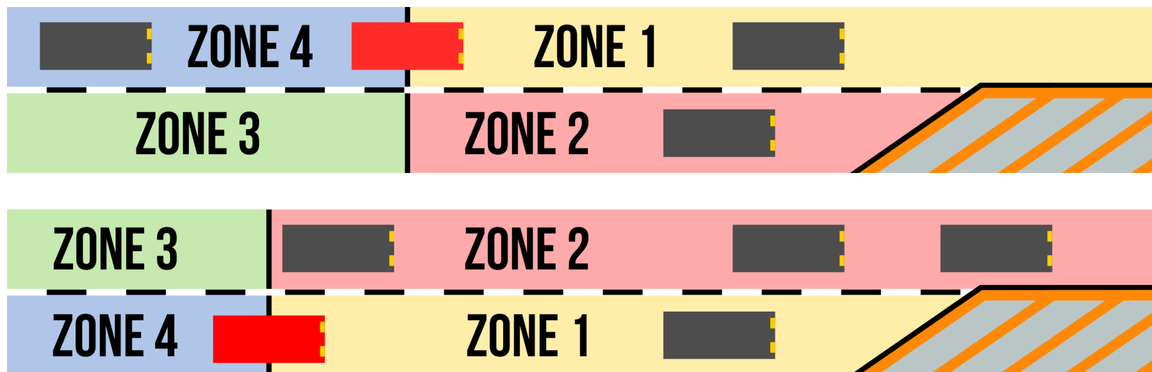


Figure 2: Illustration of zone diagram of possible areas to move to.

Our solution was to generalize the road as a series of congestion differentials across different parts of the road. At the beginning of the logic, a test driver splits the road into four distinct zones, one directly in front of itself, the second in front of itself but in the other lane, a third behind itself and in the opposing lane, and a fourth directly behind itself. Diagrams of these zones for two different test drivers in different positions on the road are provided below:

The test driver computes congestion values in each of the zones, without considering itself, and then computes an overall congesting index for the entire road. Now, in order to balance the congestion values, the test driver needs to find which zone it should move to in order to get all of the congestion values as close as possible. This is represented mathematically by our logic by summing up the differences between the overall congestion and the congestion in each of the individual zones:

$$\sum_{n=1}^{4} |C_n - C_T|$$

The test driver adds itself to each of the zones and then determines which move minimizes the sum previously mentioned. Then, the test driver will adjust its course accordingly to move into the optimal zone, thus equalizing the congestion throughout the road and then moving to a space where it will be able to move faster. In the case where the overall congestion differential between the area around the test driver and the total road is the smallest differential, the test driver will not adjust its course. Otherwise, it may decide to shift forward by speeding up, shift to the other lane by attempting to merge, shift back by slowing down, or shift backwards and try to merge at the same time.

In a real life scenario, a driver would be able to implement our Congestion Balanc-

ing Strategy, since it is possible for one to holistically determine a congestion value for each of the four general zones around ones car based upon visual cues such as the number of cars, length of the road that is accessible, and space available. Then, one would also be able to determine the area of least congestion and shift to that position, thus relieving the congestion in the portion that they used to take up and also balancing the overall congestion in the road. By developing such a strategy and defining it mathematically and logically, we were able to make a logical strategy by which one driver would be able to positively impact the overall state of a merging road that is applicable both to our model and to a real world scenario.

3.5.4 3 Lanes to 2 Lanes

Alterations Made to Model This model contains three lanes merging to form two lanes. The organization of this model is similar to Model 1, except that the implementation of the road contains an extra lane. The left-most lane contains a roadblock, which is analogous to construction being conducted in that lane, the lane completely disappearing because of a merge, or any other event causing a lane to be unavailable. In this model, vehicles are not able to begin in all three of the lanes, and our test driver is placed in the right-most lane.

With three lanes, the options for merging increase, because, at any lane, you have the option to switch to two other lanes. This is especially prevalent in the middle lane, which is neighbored by two possibly empty lanes to which you can drive. The drivers need to make a decision in this scenario to switch to the right or left, which is implemented in our model by checking the lane which does not have a roadblock first.

The test driver now needs to be aware of six different zones, as opposed to four in the previous version of this model. The relative congestion of each of these zones is calculated, and the minimum is chosen to be the optimal position to move. The complexity of this model increases greatly when the number of lanes is increased, because of the number of zones that need to be taken into account, and the number of possibilities for merging.

3.5.5 3 Lanes to 1 Lane

Alterations Made to Model This model contains three lanes merging to form one lane. This model builds off of Model 2.1, except that it adds an extra roadblock, signifying that two lanes are blocked. This restricts traffic flow significantly, making its design somewhat similar to the first model. The merging decision for the middle lane is made differently in this model than the previous one. The driver chooses to merge to the lane without a roadblock, and if that is not possible, attempts to merge

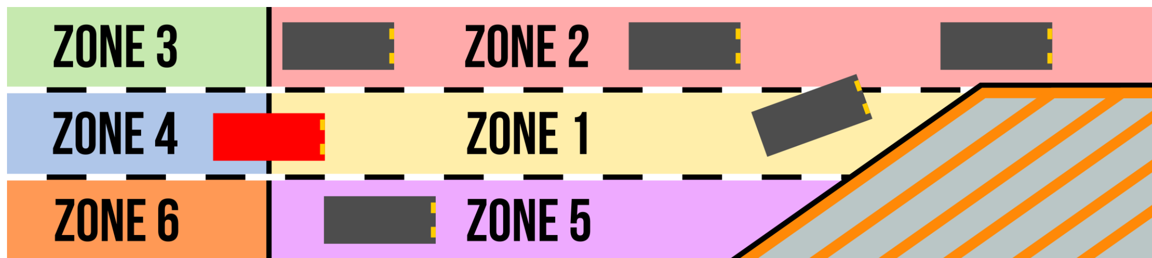


Figure 3: Modified zone diagram for a three-lane scenario.

into the other lane. The calculation for the test drivers optimal zone is implemented similarly, except it needs to take into account the bound of the second roadblock. The following diagram demonstrates this model:

4 Model Results

4.1 Test Driver (Sedan)

For a simple merge scenario where 10 sedans are present across two lanes and are all trying to merge into a single lane over the course of 2500 meters, there was a clear advantage for the entire road if even one driver used our Congestion Balancing Strategy. Two sets of five simulation runs were completed and the average summary statistics are included below:

Case	Average Drive Time (seconds)?	Average Time per Vehicle (seconds)	Average Anger Index
Without Test Driver	2350.8	574.2	0.841
WithTest Driver	2990.0	714.4	0.531

Based on this data, the test driver can be seen having a distinct impact on the average anger index, meaning that a single driver choosing to use our Congestion Balancing Strategy made a significant impact (almost -40%) on reducing road rage in this instance. While the average time per vehicle and the average drive time were increased, the values are still within 20% of each other, meaning that any increase in time caused by the presence of the test driver is much less meaningful when compared to the reduction in the anger index.

4.2 Speed Limits

The speed limit of the road where the merge is taking place also impacts what the effect of our strategy is. In a simple merge scenario where 10 sedans are present across two lanes and are all trying to merge into a single lane over the course of 2500 meters, but where there was a speed limit of 65 miles per hour rather than 35 miles per hour, the test driver was even more efficient and helped the other cars get through the merge quickly. Two sets of five simulation runs were completed and the average summary statistics are included below:

Case	Average Drive Time (seconds)?	Average Time per Vehicle (seconds)	Average Anger Index
Without Test Driver	452.2	190.9	9.79
WithTest Driver	828.2	175.7	9.02

It can be seen from this data that the test driver had a significantly larger impact on the traffic flow when the speed limits were increased. Here, when the test driver was present, all of the cars were able to individually get through quickly, however the space created by the test driver likely also caused the average drive time to be significantly increased as well. The anger indices of both cases were roughly the same which implies that as speed increases the act of creating space does not do much to reduce congestion but rather allows vehicles to pass through the merge more readily.

4.3 Sign Distance

In a simple merge scenario with a sign placed at 1500 meters away rather than 2500 meters, 10 sedans across two lanes all trying to merge into a single lane at a speed of 35 miles per hour had significantly less time to react to the upcoming merge and change lanes. The presence of the test driver once again helped alleviate the overall anger amongst the drivers in this situation yet still increased the time spent by each vehicle by a small degree. Two sets of five simulation runs were completed and the average summary statistics are included below:

Case	Average Drive Time (seconds)?	Average Time per Vehicle (seconds)	Average Anger Index
Without Test Driver	689.4	421.9	5.21
WithTest Driver	1832.2	542.5	3.18

The significant increase in average drive time yet similar value of average time seen with the presence of the test driver implies that the spacing created by the test driver was used efficiently by each of the drivers on the road; while they collectively took longer to get through, each individually was able to get through in roughly the same time and experienced a significantly lower anger index while doing so.

4.4 Test Truck

The model also allows for a driver type to be operating a different kind of vehicle, so for the following tests we ran five simulations assuming a test driver in a sedan and then five more assuming a test driver in a truck instead. The summary statistics data from each set of runs is included below:

Case	Average Drive Time (seconds)?	Average Time per Vehicle (seconds)	Average Anger Index
Test Sedan	2990.0	714.4	0.53
Test Truck	2399.0	595.4	2.09

A truck running the test driver logic is able to reduce the average drive time and the average time per vehicle, but significantly increased the average anger index. We believe that trucks, since they take up four times as much space as sedans, are able to coerce the other drivers on the road to positions where they are able to merge more easily, which makes the average times lower. However, the truck merging lanes and going near other cars skews the average anger of the other drivers also because of the size of the truck.

4.5 Multiple Test Drivers

Our model allowed for additional test drivers, which conduct our optimal strategy to increase efficiency and fairness on the street. More than one test driver would increase the number of vehicles following our strategy, meaning that every driver

wants to move into the zone with least congestion. By adding more than one test driver, one would assume that both efficiency and fairness would increase.

Case	Average Drive Time (seconds)	Average Time per Vehicle (seconds)	Average Anger Index
One Sedan	230.4	104.5	8.0
Two Sedans	1152.2	429.7	3.39

This doesnt seem to be the case. The average anger index of all the vehicles did decrease significantly, implying that the more sedans on the road, the more fair each driver is treated. Contrary to our assumption, the efficiency decreased greatly. This is because of the two sedans decisions conflicting, because they both use the same thought process. Our decision making process for the sedans emphasizes manners, allowing others to go before them if it results in an overall increase in efficiency. The more sedans on the road, the more courteous vehicles there are, decreasing the overall speed of the vehicles, even though not many specific people are angry. Our test drivers dont have too much of a motivation to get through the road faster than everyone else, so they choose to be more fair instead.

4.6 3 Lanes to 2 Lanes

Model 2.1 represents three lanes merging to two lanes. The model demonstrates the same number of vehicles distributed over the three lanes. We ran five simulations for each of the following scenarios, one without a test driver and one with a test driver. Having an extra lane allows for more space for the vehicles to maneuver and switch lanes. In this model, the test driver was improved to switch more efficiently between three lanes, increasing the optimization provided by its personalized logic. Its logic maneuvers the vehicle more often than previous models, due to the increased number of choices.

Case	Average Drive Time (seconds)	Average Time per Vehicle (seconds)	Average Anger Index
Without Test Driver	876.4	233.5	7.24
With Test Driver	425.2	118.1	5.92

Having a test driver clearly improves the efficiency and fairness of the system. Using this model versus the initial model improves the overall efficiency ten-fold (in

comparison to the data for one test driver driving a sedan). This is because there are now two lanes which are not blocked., resulting in a more even distribution of vehicles. The anger indices increased though, because of the commotion involved with switching multiple lanes, which occurs often in this model.

4.7 3 Lanes to 1 Lane

Model 2.2 represents an extension of model 2.1, except that the three lanes merge into one. This allows the vehicles to have the functionality of switching lanes often to improve efficiency, while still converging to one lane, which should cause congestion. These effects seem to balance each other out.

Case	Average Drive Time (seconds)	Average Time per Vehicle (seconds)	Average Anger Index
Without Test Driver	420.0	201.13	8.0
With Test Driver	238.4	104.9	3.91

Having three lanes as opposed to two lanes merging to one lane increases the efficiency of the model significantly. The average drive times and time per vehicle have decreased substantially, due to the increased switching between lanes. The average anger index without a test driver, on the other hand, increased, possibly because of the bottleneck effect in this scenario, in comparison to model 2.1, which merged into two lanes. The average anger index with a test driver though, decreased, because of the increased space for the test driver to move, and sway the traffic flow in a beneficial fashion.

5 Implications of Model

5.1 Congestion Balancing Strategy

The strategy that we developed to be able to balance congestion within the road was made to be applicable to a person actually driving on the road. The logic involved and the decisions that need to be made are all those that could be made by a human in a split second while driving on a highway and attempting to determine the best way to maneuver around other vehicles to get through a merge in the most efficient way possible without angering other drivers.

In the context of two lanes merging into one, our Congestion Balancing Strategy showed significant decreases in the overall anger index, but also slightly increased the time spent by each vehicle while in the simulation. This implies that the strategy helps balance the congestion in the road, however the time spent balancing the congestion does not go towards getting out of the merge in the first place. For this reason, if the vehicle employing the Congestion Balancing Strategy had been present for some time, it would likely have already been able to create an environment amongst the vehicles around itself that would be very conducive to going through the merge without any issues or a high likelihood of a crash taking place.

The Congestion Balancing Strategy works significantly better when applied in a situation where there are more than two lanes involved. In cases where there were cars merging from three lanes into either two or one lane, we noticed that the presence of the test driver would cause the time spent in the merge, average time per vehicle, and the anger index would all decrease. This implies that the strategy had a great impact on the spacing of the other cars on the road and was able to coerce them into arranging them in a way that allowed them to zipper together and merge very easily. Based off of these results, we can determine that our Congestion Balancing Method works best in scenarios where there is a lot of space to work with, since that allows the test vehicle to move to a location with a significantly lower congestion value than the overall congestion value. As such, it is clear that the Congestion Balancing Strategy would be extremely effective in highway scenarios, where three lanes and more are commonplace.

In addition to scenarios with more lanes, it was also observed that when multiple drivers implemented our Congestion Balancing Strategy the effect caused would be significantly increased. This means that if new drivers are taught our strategy of lane management then we would see lower occurrence rates of road rage, and people would generally be able to make their way through merges more efficiently. In sum, the Congestion Balancing Strategy would be an optimal way to reduce anger on the roads and allow everyone to have a more pleasant and

5.2 Efficiency vs. Fairness

Every simulation of the lane merger process outputs two very important numbers, among other indicators. These two numbers are the average amount of time it took a car to travel the 3 km of road and the average change in anger per second. These two numbers correspond to our measures of efficiency and fairness respectively. An efficient way for drivers to behave is when drivers are able to get through the 3 km of road as quickly as possible. The fairest way is when all drivers go through a similar experience. Because anger is a calculation based on a drivers relative congestion to other drivers, a lower average change in anger per second indicates that drivers during

that simulation were experiencing very similar situations. Thus, there is a difference between the most efficient and most fair way in the way that they are geared to achieve two different goals. One aims to get cars to go quickly through the 3 km. The other aims to get cars to experience similar situations, whether good or bad, through the 3 km. However these two ways may be the same. The fairest way might be the most efficient way. This is something we considered and analyzed.

The fairest way to behave during a lane merger process was determined to be our Congestion Balancing Strategy, since it consistently reduced the average anger index of each simulation run that a test driver was present in. By creating space and allowing other drivers to advance to positions where they were not affected by as much congestion, the test driver was able to decrease the difference between the congestion at any particular drivers location and the overall congestion. This in turn means that the anger index felt by any individual driver would be lower than otherwise, which implies that they believe that they received fair treatment as they progressed through the merge. In addition, the fact that the overall times for going through the 3km when a test driver is present is larger yet the average value is similar or lower implies that the actions of the test driver are spacing out the remaining cars and helping keep them far away from each other. This means that the cars that were in the front to begin with will go through the merge first and will have spent less time on the road compared to those in the back. This inherent fairness is another reason that the Congestion Balancing Strategy that we developed, even when employed by only one driver on the road, gets drivers through a merging situation in the most optimal way. Thus, since the Congestion Balancing Strategy served to get everyone through the merge process without experiencing high anger index values and preserves the initial order of vehicles, it is the most fair method of traversing a lane closure.

While fairness is an apt measure of the merits of a strategy to traverse merges, it is also important to consider the efficiency of the strategy as well, that is how long it takes for a certain number of vehicles to pass through a set distance on either side of the merge. Based on our analysis of the models results, any one drivers actions do not have a great impact on the total time spent in the merge scenario. However, if all of the drivers act in a certain way, then it is possible to increase or decrease the time required to go through the lane change. Because of this, our Congestion Balancing Strategy is not very effective on its own as it is employed by one person. In addition, this yields any single person strategy of crossing the merge pointless as well, since a larger portion of drivers acting in a certain way is necessary to heavily impact the overall times. Because of this, we can adjust the meaning of efficiency to consider only the time spent by a single individual going through the merge. In that case, our Congestion Balancing Strategy is extremely efficient, since it directly tells a driver to move to areas of lower congestion where it will be able to speed up and pass other cars. This will directly decrease the time spent by that individual in the merge

scenario. As such, the efficiency of that particular individual would be significantly increased through the use of our Congestion Balancing Strategy.

5.3 Trucks on Highways

Trucks have a clear effect on efficiency and fairness of lane merges on the highway. Implementing the Darcy-Weisbach fluid logic into a truck driver allowed drivers to get through the lane merge 10 minutes faster than a sedan driver that follows the Darcy-Weisbach fluid logic. Truck drivers also seemed to increase the average anger index of drivers. But, if we take a look at the anger index for a 65 mph highway, it is already so high at around 9. The extreme efficiency benefits of introducing trucks certainly outweigh the minor drawbacks in regards to the anger index.

The effects of implementing truck drivers on the highway is logical based on the sheer size of the truck. The large size of the truck contributes to boosting efficiency since its movement to the area of lowest congestion dramatically changes congestion throughout the road. The area that the truck left now has a low density and the area that the truck is at now will have a high density, because the truck alone already takes up so much space. This also explains the increase in anger index. Having a truck in your area congests your area while completely freeing up the areas around you. This would then frustrate drivers.

5.4 Sign Distance

Sign distance is a key contributing factor, since one must be aware of an upcoming lane closure in order to consciously act to ensure a smoother lane merger. Based on our simulations, it is clear that the sign distance must be great enough for drivers to have time to react. At the same time, there is a clear upper bound to which the sign can be placed. If you place it too early, there is a greater likelihood for other signs to be placed in between your sign and the actual location of the closure. Not only that, but it is also more likely that drivers forget about the lane closure sign, especially if there are other signs present between the lane closure sign and the lane closure. Using our simulation, we determined that placing a lane closure sign 1.5 miles before the actual lane closure in a 35 mph road results in an optimal balance between time given to react and distance to actual closure. Extrapolating this statistic to fit other speed limits, we calculated that the best location to place a sign is 150 seconds away from the actual lane closure. That is, given a car driving the speed limit, a sign should be placed a certain distance away such that if this car continues driving the speed limit, the driver will reach the lane closure in 150 seconds.

5.5 Game Theory Implications

The congestion balancing strategy mentioned above is one of many possible strategies a driver can employ in this situation. Since there are many possible strategies and clear desired outcomes, we can utilize game theoretical principles to analyze our model. Each driver knows the past behavior of nearby cars, but it does not know how the other cars will act next. If the driver utilizes the congestion balancing strategy, which we have shown to be very effective, and nearby drivers also utilize the same strategy, then the system enters what in game theory is known as Nash equilibrium. In Nash equilibrium, each driver would gain the most advantage from continuing to pursue the given strategy, and the implications of Nash equilibrium ensure that this provides the greatest overall benefit for the system. Thus, this reinforces our assertion that the more drivers that adopt our strategy, the more effective it is not only for a given driver but for the system as a whole.

5.6 Guidelines and Signage

Based on our results and analysis, we would make the following guidelines to be included in the Department of Motor Vehicles (DMV) drivers education material. Specifically, the following information would most likely be placed in a drivers manual. *Lane merges are one of the most difficult situations to handle as a driver. Everyone just wants to exit as soon as they can, which causes competition between drivers and actually slows down the process. With this thinking, drivers also get frustrated when they do not exit as soon as they wanted to. Understanding what to do in this situation requires understanding congestion. Congestion is a measure of how inefficient traffic flow is in a certain stretch of road. Although congestion is a measure, it is something that you can gauge without physically measuring anything! So, when you see a sign indicating a lane closure ahead, become more aware of your surroundings and look for two specific things, which will help you gauge congestion. These two things are how fast are the cars moving in the areas around you and how much space is there in the areas around you. The best course of action to take once you acknowledge a lane closure is coming ahead is to go to the areas where there is space and where the cars are moving slowly. This not only benefits you, but it also benefits the drivers around you as well. By following this course of action, you are able to advance more and more, as you are filling in the space around you. You also allow the drivers around you to advance more and more, because by you moving, you create space for them to advance too. With you and and the cars around you able to advance, you will exit the lane closure in a much shorter time. Because you and the cars around you will be able to exit the lane closure in a much shorter time, all of you will also be less prone to road rage. Following this course of action also helps you be a safer driver. Because*

you are moving to an area with a lot of space and slow drivers, it is very unlikely that you crash. There is plenty of space and time to make the necessary merge or acceleration that you need to.

Besides updating drivers education in this manner, we would also update the Department of Highway Safety with the following guidelines regarding sign placement. *Signs should be placed such that a driver driving constantly at the speed limit has 150 seconds from the instance of seeing the sign to the instance he passes what the sign is referring to. For example, a sign indicating a lane closure ahead should be placed 1.5 miles before the actual closure to give a driver going the speed limit of 35 mph 150 seconds to continue driving at 35 mph before reaching the lane closure. The corresponding distances for this 150 second time threshold are the shortest distances that allow everyone to merge into the other lane as efficiently and fairly as possible. Decreasing this distance limits the time drivers can react to the sign intelligently. If you increase this distance, you run into the risk of other obstructions occurring between the sign and the closure, which will certainly cause confusion.*

6 Strengths and Weaknesses

6.1 Strengths

1. From a generalized situation, our model makes few assumptions that werent directly stated in the problem. The distance of the sign, the types of drivers and their behavior, and our given drivers strategy were all left as variables that were modified to test different situations. Our lack of assumptions meant we could analyze the general case, which works well as we dont know information about the specific case.

2. Our model is versatile and robust, able to simulate a variety of driving scenarios. If, for example, we were given specific information about sign distance, placement of cars, or nature of traffic flow obstruction, we could easily implement that information to model the specific scenario and determine the optimal strategy and overall runtime of that scenario. In addition, the simulation we created can be more widely applied, as the way we construct our road, drivers, and signs do not limit our model from simulating only a two-lane, one-exit scenario. In fact, we easily converted our model to test the different situations of three lanes and different speed limits.

3. The logical strategy our model created can actually be employed by drivers in real-life scenarios. While driving in real life, drivers can actually qualitatively

gauge congestion, based on free space, and take the necessary measures to balance it — which is the underlying behavior of our congestion balance equation that guides our strategy. While our equation is based on math, any driver can intuitively follow our principles without needing to implement complex formulas or algorithms to decide what to do next.

4. Since our model is a simulation, we can easily test many scenarios that are unfeasible to test in real life. Obviously, it is impractical to hire a test driver to drive through a lane closure area hundreds of times in order to empirically determine the results of different strategies. However, our model can perform these experiments, potentially under circumstances that are impossible to test in real life. We can retain the accuracy of experimentation through our simulation and not resort to purely mathematical constructs which may not be realistic. Our simulation will be, however, as it is based off of everyday driving logic and behavior. Furthermore, we can test changes in initial variables and employ a Monte Carlo approach.

6.2 Weaknesses

1. We assume that all drivers can be permanently classified into one of our six categories. This compartmentalizes driving styles to make our problem feasible; however, it may not be completely accurate. For example, a person may be a safe driver, but when he sees an opportunity to pass other cars, he will immediately take it, making him a selfish driver under certain circumstances. Overall, our simulation gives a good distribution of generalized drivers one may encounter, without worrying about the specific, unique people in real life.

2. We assume that car types can also be classified into one of four categories, with generalized attributes for each, with special focus on length. However, cars on the road may have a myriad of different sizes and characteristics.

3. Our model does not produce a definite action to take at any time, but rather analyzes the results of performing the possible actions at that time. It is up to the driver to decide which action to perform. For example, the model may reveal that merging lanes would slightly reduce congestion, but a driver may know that merging may potentially result in a crash for a crowded lane. Therefore, the driver may choose not to follow the optimal action sequence, but rather the one that would prevent him from crashing.

4. Since each situation is different in terms of drivers and road conditions, the model must be run a large number of times to receive good results of overall

behavior and generalizable patterns. This requires a longer runtime to produce accurate results, because each simulation takes a while to run to completion.

7 Conclusion

The results from our model indicate that balancing congestion is the most important factor in ensuring optimal traffic flow through a lane closure area. If a given driver analyzes the options open to him and takes the one most likely to balance congestion in the entire system, not only does he get through the section of road more efficiently, but every driver also experiences less overall anger and everyone gets through more effectively and safely. If every driver employs the strategy to balance congestion, the system as a whole can be near optimally balanced and will reach max efficiency, getting everyone through in the shortest amount of time possible. We determined that the balancing congestion strategy is best regardless of two lanes or three lanes, a 35 mph or 65 mph speed limit.

Although overall driver anger can be greatly reduced with just one congestion-conscious driver, making the system more fair, the efficiency of the system cannot be greatly increased unless every driver follows similar principles. The zipper method, where drivers stay in their lanes until the very end of the merging stretch, is an example of a near optimal method of balancing congestion that does not work unless everyone follows it. Although certain governments are proponents of this method, it cannot be effective, increasing fairness and efficiency, unless all drivers follow it. In contrast, we have developed a strategy for balancing congestion that one individual can employ himself, that will increase not only his own efficiency but decrease the anger of the entire system. Through mass education and outreach, we believe that governments can make the first step in ensuring optimal efficiency for all – if everyone follows our strategy. We believe that the large incentive to adopt our method both for personal gain and societal well-being will ensure that our method receives widespread adoption.

8. References

Bertucci, A. (2006). Sign Legibility Rules of Thumb. Retrieved November 7, 2015, from http://www.ussc.org/SignLegibilityLettersize.pdf

Bianchi, A., & Summala, H. (2004). The "genetics" of driving behavior: Parents' driving style predicts their children's driving style. *Accident Analysis & Prevention, 36*(4), 655-659. doi:10.1016/S0001-4575(03)00087-3

Hearfield, J. (2012). Water Flowing in Pipes. Retrieved November 7, 2015, from http://www.johnhearfield.com/Water/Water_in_pipes.htm

Laing, C. (2010). Aggressive Driving. Retrieved November 7, 2015, from http://www.popcenter.org/problems/aggressive_driving/

MAG Internal Truck Travel Survey and Truck Model Development Study. (2007, December 1). Retrieved November 7, 2015, from http://www.camsys.com/pubs/TRANS_2011-02-25_mag-internal-truck-travel-survey-and-truck-model-development-study.pdf

Manual on Uniform Traffic Control Devices for Streets and Highways. (2012, May 1). Retrieved November 7, 2015, from http://mutcd.fhwa.dot.gov/pdfs/2009r1r2/mutcd2009r1r2edition.pdf

Nash, John (1951) "Non-Cooperative Games" The Annals of Mathematics 54(2):286-295.

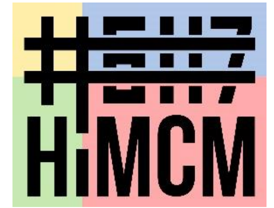
P., T. (2010, August 26). Jam yesterday, jam tomorrow. Retrieved November 7, 2015, from http://www.economist.com/blogs/banyan/2010/08/great_chinese_traffic_jam

Sansone, R. A., & Sansone, L. A. (2010). Road Rage: What's Driving It? *Psychiatry (Edgmont)*, *7*(7), 14–18.

Schroeder, P., Kostyniuk, L., & Mack, M. (2013, December). *2011 National Survey of Speeding Attitudes and Behaviors*. (Report No. DOT HS 811 865). Washington, DC: National Highway Traffic Safety Administration

Wittwer, J.W., "Monte Carlo Simulation Basics" From Vertex42.com, June 1, 2004, http://www.vertex42.com/ExcelArticles/mc/MonteCarloSimulation.html

Zipper Merge. (n.d.). Retrieved November 7, 2015, from http://www.dot.state.mn.us/zippermerge/



November 8, 2015

The Honorable Anthony Foxx
Secretary
U.S. Department of Transportation
1200 New Jersey Ave, SE
Washington, DC 20590

Dear Secretary Foxx,

As you well know, traffic jams are a growing problem on our nation's roads and highway systems. One of the major causes of traffic blockages are short-term construction or maintenance projects that result in the temporary closure of a lane. When a formerly two-lane roadway must merge into one lane, the cars on it must slow down greatly, increasing traffic congestion, decreasing safety, and contributing to potential road rage.

As members of the team chosen to analyze these lane closure scenarios, we have dedicated ourselves to testing different strategies to bypass a lane closure and determine the optimal strategy, both on the individual and systemic level. In other words, we analyzed both the strategies that any given driver can employ in order to increase traffic flow and his own speed, as well as overall optimal strategies that can be implemented on a larger scale. If an individual driver follows a strategy of balancing congestion in the system – that is, moving or switching to less congested areas – we found that his average time to pass the closure can be decreased by 8%, and the overall anger of all the drivers on the road can be decreased by 10%. The guidelines we have created are as follows, and additional detail and rationale are included in our complete paper.

Guidelines

1. Notice congestion, a measure of inefficient traffic flow, and seek to switch to the less congested road.
2. Fill in open space if present in other lane to balance congestion and open up space for cars behind you.
3. Merging is not necessary until 3 seconds before the beginning of the lane closure.
4. Signs indicating lane closure should be optimally placed, based on speed limit, 150 seconds before lane closure.

Through widespread educational measures, such as inclusion of our guidelines in the Department of Motor Vehicles driver education materials, the strategy we found can be implemented on a larger scale, resulting in lower driver anger, improved traffic flow, and hopefully fewer traffic jams. If lane closure signs are placed at our optimally determined distance, drivers will have enough time to switch lanes safely and efficiently, again resulting in smoother traffic flow. We implore you to implement our guidelines to improve traffic efficiency and continue advancing

the core mission of the U.S. Transportation system – to get people from one place to another as fast and safely as possible.

Respectfully,

Team 6117

Team 6117

9. Appendix

Implementation of Model (Code)

CODE

Model 1:

Driver

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileOutputStream;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;

public class Driver {
	private static PrintWriter file; // for sets of tests
	private static FileWriter masterFile; // for overall analysis
of data
	private static BufferedReader reader;

	private static int num_tests = 2;

	public static int num_cars = 0;
	private static int num_sedans = 10;
	private static int num_vans = 0;
	private static int num_trucks = 0;

	// dont change these values
	private static int num_random_drivers = 0;
	private static int num_aggressive_drivers = 0;
	private static int num_conservative_drivers = 0;
	private static int num_slow_drivers = 0;
	private static int num_fast_drivers = 0;
	private static int num_safe_drivers = 0;
	private static int num_selfish_drivers = 0;
```

```
	private static int num_fair_drivers = 0;
	private static int num_test_drivers = 0;

	private static String crash_result = "";

	public static void main(String[] args) throws IOException {
		// each set of tests gets printed out in file

		String text = "";
		try {
			reader = new BufferedReader(new 
FileReader("master_file.csv"));
			try {
			   StringBuilder builder = new StringBuilder();
			   String line = reader.readLine();

			   while (line != null) {
			       builder.append(line);
			       builder.append(System.lineSeparator());
			       line = reader.readLine();
			   }
			   text = builder.toString();
			} finally {
			   reader.close();
			}
		} catch(Exception e) {System.out.println("master file read 
error");}

		String temp = "";
		file = null;
		for (int i=1111; i<9999; i++) {
			if (!text.contains(""+i)) {
				temp = ""+i+".csv";
				break;
			}
		}
		try {
			file = new PrintWriter(temp);
		} catch (Exception e) {System.out.println("file writing 
error");}
		try {
			masterFile = new FileWriter("master_file.csv", true);
```

```
            } catch (Exception e) {System.out.println("master file
write error");}

            // masterFile.append("TEST FILE,# CARS,# SEDANS,# VANS,#
TRUCKS,# RANDOM DRIVERS,# AGGRESSIVE DRIVERS,# CONSERVATIVE DRIVERS,#
SLOW DRIVERS,# FAST DRIVERS,# SAFE DRIVERS,# SELFISH DRIVERS,# FAIR
DRIVERS,# TEST DRIVERS,AVG. TOTAL RUNTIME,AVG. AVG. TIME/CAR,#
CRASHES\n");

            /* SET UP TESTS */

            num_cars = num_sedans + num_vans + num_trucks;
            int total_runtime = 0;
            double avg_time_per_car = 0;

            // beginning of test
            file.println("TOTAL RUNTIME,AVG TIME PER CAR,CRASHED,AVG
ANGER");

            num_cars = num_sedans + num_vans + num_trucks;

            double avg_runtime = 0.0;
            double avg_avg_time_per_car = 0.0;
            int num_crashes = 0;

            for (int i=0; i<num_tests; i++) {

                  Road r = new Road();

                  //r.createVehicle("sedan", "test", 5, 2);

                  int pos;
                  int lane;
                  for(int j=0;j<num_sedans;j++){
                        pos = (int)(Math.random()*1500)+15;
                        lane = (int)(Math.random()*2)+1;
                        System.out.println(pos + " " + lane);
                        int driverType = (int)(Math.random()*7)+1;
                        switch(driverType) {
                        case 1:
                              if (!r.createVehicle("sedan",
"aggressive", pos, lane)) {
                                    j--;
```

```
                    }
                    num_aggressive_drivers++;
                    System.out.println("created aggressive");
                    break;
                case 2:
                    if (!r.createVehicle("sedan",
"conservative", pos, lane)) {
                        j--;
                    }
                    num_conservative_drivers++;
                    System.out.println("created
conservative");
                    break;
                case 3:
                    if (!r.createVehicle("sedan", "fair", pos,
lane)) {
                        j--;
                    }
                    num_fair_drivers++;
                    System.out.println("created fair");
                    break;
                case 4:
                    if (!r.createVehicle("sedan", "fast", pos,
lane)) {
                        j--;
                    }
                    num_fast_drivers++;
                    System.out.println("created fast");
                    break;
                case 5:
                    if (!r.createVehicle("sedan", "safe", pos,
lane)) {
                        j--;
                    }
                    num_safe_drivers++;
                    System.out.println("created safe");
                    break;
                case 6:
                    if (!r.createVehicle("sedan", "selfish",
pos, lane)) {
                        j--;
                    }
                    num_selfish_drivers++;
```

```
                                        System.out.println("created selfish");
                                        break;
                              case 7:
                                        if (!r.createVehicle("sedan", "slow", pos,
lane)) {
                                                  j--;
                                        }
                                        num_slow_drivers++;
                                        System.out.println("created slow");
                                        break;
                              default:
                                        if (!r.createVehicle("sedan", "random",
pos, lane)) {
                                                  j--;
                                        }
                                        num_random_drivers++;
                                        break;
                              }
                    }

                    for(int j=0;j<num_vans;j++){
                              pos = (int)(Math.random()*1500)+15;
                              lane = (int)(Math.random()*2)+1;
                              System.out.println(pos + " " + lane);
                              int driverType = (int)(Math.random()*7)+1;
                              switch(driverType) {
                              case 1:
                                        if (!r.createVehicle("van", "aggressive",
pos, lane)) {
                                                  j--;
                                        }
                                        num_aggressive_drivers++;
                                        break;
                              case 2:
                                        if (!r.createVehicle("van",
"conservative", pos, lane)) {
                                                  j--;
                                        }
                                        num_conservative_drivers++;
                                        System.out.println("created
conservative");
                                        break;
                              case 3:
```

```
                                if (!r.createVehicle("van", "fair", pos,
lane)) {
                                      j--;
                                }
                                num_fair_drivers++;
                                System.out.println("created fair");
                                break;
                          case 4:
                                if (!r.createVehicle("van", "fast", pos,
lane)) {
                                      j--;
                                }
                                num_fast_drivers++;
                                System.out.println("created fast");
                                break;
                          case 5:
                                if (!r.createVehicle("van", "safe", pos,
lane)) {
                                      j--;
                                }
                                num_safe_drivers++;
                                System.out.println("created safe");
                                break;
                          case 6:
                                if (!r.createVehicle("van", "selfish",
pos, lane)) {
                                      j--;
                                }
                                num_selfish_drivers++;
                                System.out.println("created selfish");
                                break;
                          case 7:
                                if (!r.createVehicle("van", "slow", pos,
lane)) {
                                      j--;
                                }
                                num_slow_drivers++;
                                System.out.println("created slow");
                                break;
                          default:
                                if (!r.createVehicle("vans", "random",
pos, lane)) {
                                      j--;
```

```
                    }
                    num_random_drivers++;
                    break;
               }
          }

          for(int j=0;j<num_trucks;j++){
               pos = (int)(Math.random()*1500)+15;
               lane = (int)(Math.random()*2)+1;
               System.out.println(pos + " " + lane);
               int driverType = (int)(Math.random()*7)+1;
               switch(driverType) {
               case 1:
                    if (!r.createVehicle("truck",
"aggressive", pos, lane)) {
                         j--;
                    }
                    num_aggressive_drivers++;
                    break;
               case 2:
                    if (!r.createVehicle("truck",
"conservative", pos, lane)) {
                         j--;
                    }
                    num_conservative_drivers++;
                    System.out.println("created
conservative");
                    break;
               case 3:
                    if (!r.createVehicle("truck", "fair", pos,
lane)) {
                         j--;
                    }
                    num_fair_drivers++;
                    System.out.println("created fair");
                    break;
               case 4:
                    if (!r.createVehicle("truck", "fast", pos,
lane)) {
                         j--;
                    }
                    num_fast_drivers++;
                    System.out.println("created fast");
```

```
                              break;
                         case 5:
                              if (!r.createVehicle("truck", "safe", pos,
lane)) {
                                   j--;
                              }
                              num_safe_drivers++;
                              System.out.println("created safe");
                              break;
                         case 6:
                              if (!r.createVehicle("truck", "selfish",
pos, lane)) {
                                   j--;
                              }
                              num_selfish_drivers++;
                              System.out.println("created selfish");
                              break;
                         case 7:
                              if (!r.createVehicle("truck", "slow", pos,
lane)) {
                                   j--;
                              }
                              num_slow_drivers++;
                              System.out.println("created slow");
                              break;
                         default:
                              if (!r.createVehicle("trucks", "random",
pos, lane)) {
                                   j--;
                              }
                              num_random_drivers++;
                              break;
                         }
                    }

                    r.createVehicle("roadblock", "",
r.roadblock_position, 2);

                    // create test vehicle
                    r.createVehicle("sedan", "test", 0, 2);

                    r.orderVehicles();
```

```
			r.run();

			double avg_anger = 0;
			for(Vehicle v: r.vehicles){
				avg_anger += v.anger;
			}
			avg_anger = avg_anger/r.vehicles.size();

			total_runtime = r.total_runtime;
			avg_runtime += total_runtime;

			String crash_result = (r.crashed) ?
r.crash_vehicle.logic.toString() : ""; // TYPE OF DRIVER THAT CRASHES
			if(!r.crashed){
				avg_time_per_car = r.avg_time_per_car;
				avg_avg_time_per_car += avg_time_per_car;
			}else{
				num_crashes++;
			}

			file.println(total_runtime + "," + avg_time_per_car +
"," + crash_result + "," + avg_anger);
		}

		avg_runtime = ((double) avg_runtime)/num_tests;
		avg_avg_time_per_car = ((double)
avg_avg_time_per_car)/num_tests;
		System.out.println("--Overall Statistics--");
		System.out.println("Average Runtime:" + avg_runtime);
		System.out.println("Average Average Time/Car: " +
avg_avg_time_per_car);


		System.out.println("Total Crashes: " + num_crashes);
		masterFile.append(temp + "," + num_cars + "," + num_sedans
+ "," + num_vans + "," + num_trucks + "," + num_random_drivers + ","
+ num_aggressive_drivers + "," + num_conservative_drivers + "," +
num_slow_drivers + "," + num_fast_drivers + "," + num_safe_drivers +
"," + num_selfish_drivers + "," + num_fair_drivers + "," +
num_test_drivers + "," + avg_runtime + "," + avg_avg_time_per_car +
"," + num_crashes + "\n");

		// end of test
```

```
			masterFile.close();
			file.close();
		}

}
```

Road

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

// 2 constructors
// arguments: num of each type of car, list of vehicles

public class Road {
	public int num_cars; // number of cars
	public int speed_limit = 65; // speed limit of road
	public int road_length = 3000; // road length - in meters
	public int num_lanes = 2; // number of lanes
	public int roadblock_position = 2500; // position of road-block
from beginning of road

	public Vehicle crash_vehicle; // vehicle that caused crash

	// add up to num_cars
	public int num_sedans = 0;
	public int num_vans = 0;
	public int num_trucks = 0;

	// lanes
	public List<Vehicle> lane1 = new ArrayList<Vehicle>();
	public List<Vehicle> lane2 = new ArrayList<Vehicle>(); // has
roadblock

	// vehicle array
	public List<Vehicle> vehicles = new ArrayList<Vehicle>(); //
adds all vehicles created

	// universal time for simulation
	public int time = 0;
```

```
	public List<Double> total_congestions = new
ArrayList<Double>();

	// return information
	public int total_runtime;
	public double avg_time_per_car;

	public boolean crashed = false;

	public Road(int vehicles){
		initLanes();
		for (int i=0; i<vehicles; i++) {
			int vType = (int)Math.random()*3+1;
			String type = "";
			switch(vType){
			case 1:
				type = "sedan";
				break;
			case 2:
				type = "van";
				break;
			case 3:
				type = "truck";
				break;
			}
			int vDriver = (int)Math.random()*9+1;
			String driverType = "";
			switch(vDriver){
			case 1:
				driverType = "random";
				break;
			case 2:
				driverType = "aggressive";
				break;
			case 3:
				driverType = "conservative";
				break;
			case 4:
				driverType = "slow";
				break;
			case 5:
				driverType = "fast";
				break;
```

```
				case 6:
					driverType = "safe";
					break;
				case 7:
					driverType = "selfish";
					break;
				case 8:
					driverType = "fair";
					break;
				case 9:
					driverType = "test";
					break;
				}
				int pos = (int)Math.random()*1500+15;
				int lane = (int)Math.random()*2+1;
				createVehicle(type,driverType,pos,lane);
			}
		}

		// road constructor for given vehicle arraylist
		public Road(){
			initLanes();
		}

		public boolean createVehicle(String type, String driver, int
pos, int lane) {
			Vehicle newV = null;
			if (type.equals("sedan")) {
				newV = new Sedan();
			}
			else if (type.equals("van")) {
				newV = new Van();
			}
			else if (type.equals("truck")) {
				newV = new Truck();
			}else if(type.equals("roadblock")){
				newV = new RoadBlock();
			}
			if (!placeVehicle(newV, pos, lane)) {
				return false;
			}
			vehicles.add(newV);
			if (driver.equals("random")) {
```

```
			newV.logic = new RandomDriver(newV);
		}
		else if (driver.equals("aggressive")) {
			newV.logic = new AggressiveDriver(newV);
		}
		else if (driver.equals("conservative")) {
			newV.logic = new ConservativeDriver(newV);
		}
		else if (driver.equals("slow")) {
			newV.logic = new SlowDriver(newV);
		}
		else if (driver.equals("fast")) {
			newV.logic = new FastDriver(newV);
		}
		else if (driver.equals("safe")) {
			newV.logic = new SafeDriver(newV);
		}
		else if (driver.equals("selfish")) {
			newV.logic = new SelfishDriver(newV);
		}
		else if (driver.equals("fair")) {
			newV.logic = new FairDriver(newV);
		}
		else if (driver.equals("test")) {
			newV.logic = new TestDriver(newV);
		}
		return true;

	}

	public boolean placeVehicle(Vehicle v, int pos, int lane) {
	// places a vehicle at a specific position (based off of front
index) if the vehicle goes off the array, those elements are left out

		if (lane==1 && lane1.get(pos) == null) {
			for (int i=0; i<v.length; i++) {
				if (pos-i >= 0) {
					lane1.set(pos-i, v);
				}
			}
			return true;
		}
		else if (lane==2 && lane2.get(pos) == null) {
```

```
				for (int i=0; i<v.length; i++) {
					if (pos-i >= 0) {
						lane2.set(pos-i, v);
					}
				}
				return true;
			}
			else if (lane==3 && lane1.get(pos) == null &&
lane2.get(pos) == null) {
				for (int i=0; i<v.length; i++) {
					if (pos-i >= 0) {
						lane1.set(pos-i, v);
						lane2.set(pos-i, v);
					}
				}

				return true;
			}

			return false;
		}

		public void removeVehicle(Vehicle v) {
			while (lane1.indexOf(v) != -1) {
				lane1.set(lane1.indexOf(v), null);
			}
			while (lane2.indexOf(v) != -1) {
				lane2.set(lane2.indexOf(v), null);
			}
		}

		public boolean moveVehicle(Vehicle v, int pos, int lane) {
			removeVehicle(v);
			if (!placeVehicle(v,pos,lane)) {
				crash_vehicle = v;
				v.crashed = true;
				return false;	// crash occurred
			}
			return true;
		}

		public boolean orderVehicles() {
			// sorts vehicles in descending order
```

```
            Collections.sort(vehicles, new CustomComparator());
            return true;
      }

      public void run() {

            while (nextSecond().equals("running")) {
                  time++;
            }

            total_runtime = time;
            System.out.println("Total Runtime: "+time);

            avg_time_per_car = 0;
            for (Vehicle v: vehicles) {
                  System.out.println(v.timeOnRoad);
                  avg_time_per_car += v.timeOnRoad;
            }
            avg_time_per_car = avg_time_per_car/(vehicles.size());
            System.out.println("Avg. Time/Car: "+ avg_time_per_car);

            System.out.println("ct size: " +
total_congestions.size());

            // calculate everyone's change in anger
            for(Vehicle v: vehicles){
                  if(!(v instanceof RoadBlock)){

System.out.println(v.subset_congestions.size());
                        double delta_a = 0;
                        for(int i=0;i<total_congestions.size();i++){
                              delta_a +=
(v.subset_congestions.get(i)-total_congestions.get(i));
                        }
                        v.anger = delta_a; // more congestion, higher
anger

                        // System.out.println(v.anger);
                  }
            }
      }

      // increments through each second
```

```
	public String nextSecond(){
		orderVehicles(); // sorts vehicles in descending order

		// congestion calculations
		double avg_velocity = 0;
		for(Vehicle w: vehicles){
			avg_velocity += w.speed;
		}
		avg_velocity = ((double) avg_velocity)/(vehicles.size());
		double ct = congestion(avg_velocity, vehicles.size(),
3000, 2);

		total_congestions.add(ct);

		for (Vehicle v: vehicles) {
			// calculate Cs
			int num_cars_range = 0;
			double avg_velocity_range = 0;

			// loop range of 100 around car
			for(int
i=v.findIndex(this)-50;i<v.findIndex(this)+50;i++){
				if(i >= 0 && i < 3000){
					if(lane1.get(i) != null){
						num_cars_range++;
						avg_velocity_range +=
lane1.get(i).speed;
					}

					if(lane2.get(i) != null){
						num_cars_range++;
						avg_velocity_range +=
lane2.get(i).speed;
					}
				}
			}

			if(num_cars_range != 0){
				avg_velocity_range = ((double)
avg_velocity_range)/num_cars_range;
			}else{
				avg_velocity_range = 0;
			}
```

```
                double cs = congestion(avg_velocity_range,
num_cars_range, 100, 2);
                v.subset_congestions.add(cs);

                if (v.drive(this)) {  // if a crash took place, then
break out of the simulation
                    crashed = true;
                    return "crash";
                }
            }
            if (allCarsExited()) {
                // set anger values
                return "done";
            }
            return "running";          // implies simulation is still
running in this second
        }

        public boolean allCarsExited() {
            for (Vehicle v: vehicles) {
                if (!v.exited && !v.crashed)
                    return false;
            }
            return true;
        }

        public void initLanes() {
            for (int i=0; i<3000; i++) {
                lane1.add(null);
                lane2.add(null);
            }

        }

        public double congestion(double vel, int cars, int len, int
lanes) {
            return vel*vel*cars/len/lanes/2;
        }

        private class CustomComparator implements Comparator<Vehicle> {
           @Override
           public int compare(Vehicle v1, Vehicle v2) {
```

```
            return
Integer.valueOf(v2.findIndex(Road.this)).compareTo(Integer.valueOf(v1
.findIndex(Road.this))); // descending order
        }
    }
}
```

Vehicle

```
import java.util.ArrayList;
import java.util.List;


public class Vehicle {
     public double anger = 0;
     public int max_speed;           // the maximum speed attainable
by this kind of car (kph)
     public int speed;                    // the current speed of
this car (kph)
     public int length;                   // the length of this car
     public boolean crashed = false;
     public double brakeAccel;       // the braking acceleration of
this car
     public double accel;            // the acceleration of this car
     public int mergeTime;           // the time (in seconds) in
which this vehicle can merge
     public boolean exited = false;  // whether or not the vehicle
is still in the 3km stretch of road
     public DriverLogic logic;
     public int timeOnRoad = 0;

     public List<Double> subset_congestions = new
ArrayList<Double>();

     public Vehicle() {

     }


     public boolean drive(Road road) {    // returns whether or not
a crash took place
          if (exited) {
               return false;
```

```
		}

		if(crashed){
			System.out.println("crashed");
			return true;
		}
		logic.drive(road, this);
		return false;
	}


	public int findIndex(Road road) {

		if (road.lane1.indexOf(this) != -1) {
			return road.lane1.lastIndexOf(this);
		}
		else {
			return road.lane2.lastIndexOf(this);
		}

	}

	public int findLane(Road road) {
		int lane = 0;
		if (road.lane1.indexOf(this) != -1) {
			lane+=1;
		} if (road.lane2.indexOf(this) != -1) {
			lane+=2;
		}
		return lane;	// returns 1 if car is in lane 1, 2 if
lane 2, or 3 if in both lanes	(0 if vehicle doesn't exist)
	}

}
```

DriverLogic

```
import java.util.ArrayList;
import java.util.List;


public class DriverLogic {
```

```
	public String type;

	public int mergeStatus = 0;
	int maxTimeToImpact;
	int slowTime;
	int speedTime;



	public List<String> actions = new ArrayList<String>();

	public DriverLogic() {

	}

	public void drive(Road road, Vehicle v) {

	}

	public String toString(){
		return "driver logic";
	}

	public int findIndex(Road road, Vehicle v) {

		if (road.lane1.indexOf(v) != -1) {
			return road.lane1.indexOf(v)+v.length-1;
		}
		else {
			return road.lane2.indexOf(v)+v.length-1;
		}

	}

	public int findLane(Road road, Vehicle v) {
		int lane = 0;
		if (road.lane1.indexOf(v) != -1) {
			lane+=1;
		} if (road.lane2.indexOf(v) != -1) {
			lane+=2;
		}
		return lane;	// returns 1 if car is in lane 1, 2 if
lane 2, or 3 if in both lanes	(0 if vehicle doesn't exist)
```

```
	}

	public boolean move(int delta, Road road, Vehicle v) {
		int newPos = v.findIndex(road)+delta;
		if (newPos < 3000) {
			return road.moveVehicle(v, findIndex(road, v)+delta,
findLane(road, v));
		}
		else {
			v.exited=true;
			v.timeOnRoad = road.time;
			road.removeVehicle(v);
		}
		return false;
	}

	public double mps(int speed) {  // converts speed (kph) to m/s
		return speed*0.277778;
	}

	public void accel(Road road, Vehicle v) {
		int spacesInFront = 0;
		for (int i=findIndex(road, v)+1; i<3000; i++) {
			if (findLane(road, v) == 1) {
				if (road.lane1.get(i) == null) {
					spacesInFront++;
				} else {
					break;
				}
			}
			else if (findLane(road, v) == 2) {
				if (road.lane2.get(i) == null) {
					spacesInFront++;
				} else {
					break;
				}
			}
		}

		int timeToImpact = (int)(spacesInFront/(v.speed+.01));
		if (maxTimeToImpact - timeToImpact < slowTime) {
			v.speed = v.speed+(int)v.brakeAccel;
			if (v.speed<1) {
```

```
                v.speed = 0;
            }
            actions.add("brake");
            // System.out.println("brake");
        } else if (maxTimeToImpact - timeToImpact > speedTime) {
            v.speed = v.speed+(int)v.accel;
            actions.add("accel");
            if (v.speed>v.max_speed) {
                v.speed = v.max_speed;
            }

            if(v.speed>road.speed_limit){
                v.speed = road.speed_limit;
            }
            // System.out.println("accel");
        }
    }

    // default merge-left
    // no merge buffer
    public void mergeLeft(Road road, Vehicle v){
        if(v.findLane(road) == 1){
            return;
        }

        int pos = v.findIndex(road);

        if(mergeStatus == 0){
            for(int i=pos;i>pos-v.length;i--){
                if(i >= 0 && i < road.roadblock_position &&
road.lane1.get(i) == null){
                    return;
                }
            }

            mergeStatus++;
        }else{
            mergeStatus++;
            if(mergeStatus == 4){
                road.moveVehicle(v, pos, 1);
            }
        }
    }
```

```
	public void mergeRight(Road road, Vehicle v){
		if(v.findLane(road) == 2){
			return;
		}

		int pos = v.findIndex(road);

		if(mergeStatus == 0){
			for(int i=pos;i>pos-v.length;i--){
				if(i >= 0 && i < road.roadblock_position - 100
&& road.lane2.get(i) == null){
					return;
				}
			}

			mergeStatus++;
		}else{
			mergeStatus++;
			if(mergeStatus == 4){
				road.moveVehicle(v, pos, 2);
			}
		}
	}

	public double congestion(double vel, int cars, int len, int
lanes) {
		return vel*vel*cars/len/lanes/2;
	}
}
```

AggressiveDriver

```
public class AggressiveDriver extends DriverLogic{



	public AggressiveDriver(Vehicle v) {
		type="aggressive";
		v.speed = (int)(.6*v.max_speed);
		maxTimeToImpact = (int)Math.ceil(v.max_speed/v.accel)+1;
		slowTime = 3;
		speedTime = 8;
```

```
	}


	public void drive(Road road, Vehicle v) {

		accel(road,v);

		int delta = (int)Math.ceil(mps(v.speed))+1;
		move(delta, road, v);

		mergeLeft(road, v);
	}

	// custom merge-left
	// merge buffer of 0
	public void mergeLeft(Road road, Vehicle v){
		if(v.findLane(road) == 1){
			return;
		}

		int pos = v.findIndex(road);

		if(mergeStatus == 0){
			for(int i=pos;i>pos-v.length;i--){
				if(i >= 0 && i < road.roadblock_position &&
road.lane1.get(i) == null){
					return;
				}
			}

			mergeStatus++;
		}else{
			System.out.println(mergeStatus);
			mergeStatus++;
			if(mergeStatus == 3){
				road.moveVehicle(v, pos, 1);
			}
		}
	}
}
```

ConservativeDriver

```
public class ConservativeDriver extends DriverLogic{

	public ConservativeDriver(Vehicle v) {
		type="conservative";
		v.speed = (int)(.4*v.max_speed);	//v.max_speed);
		maxTimeToImpact = (int)Math.ceil(v.max_speed/v.accel)+4;
		slowTime = 6;
		speedTime = 14;
	}

	public void drive(Road road, Vehicle v) {

		accel(road,v);

		int delta = (int)Math.ceil(mps(v.speed))+1;
		move(delta, road, v);

		mergeLeft(road, v);
	}

	// custom merge-left
	// merge buffer of 5
	public void mergeLeft(Road road, Vehicle v){
		if(v.findLane(road) == 1){
			return;
		}

		int pos = v.findIndex(road);

		if(mergeStatus == 0){
			for(int i=pos+5;i>pos-v.length-5;i--){
				if(i >= 0 && i < road.roadblock_position &&
road.lane1.get(i) == null){
					return;
				}
			}

			mergeStatus++;
		}else{
			System.out.println(mergeStatus);
			mergeStatus++;
			if(mergeStatus == 4){
				road.moveVehicle(v, pos, 1);
```

```
                }
            }
        }
}
```

FairDriver

```
public class FairDriver extends DriverLogic     {

      public FairDriver(Vehicle v) {
            type="fair";
            v.speed = (int)(.4*v.max_speed);     //v.max_speed);
            maxTimeToImpact = (int)Math.ceil(v.max_speed/v.accel)+2;
            slowTime = 4;
            speedTime = 10;
      }

      public void drive(Road road, Vehicle v) {

            accel(road,v);

            int delta = (int)Math.ceil(mps(v.speed))+1;
            move(delta, road, v);

            mergeLeft(road, v);
      }

      // custom merge-left
      // merge buffer of 5 (reasonable)
      public void mergeLeft(Road road, Vehicle v){
            if(v.findLane(road) == 1){
                  return;
            }

            int pos = v.findIndex(road);

            if(mergeStatus == 0){
                  for(int i=pos+5;i>pos-v.length-5;i--){
                        if(i >= 0 && i < road.roadblock_position &&
road.lane1.get(i) == null){
                              return;
                        }
```

```
				}

				mergeStatus++;
			}else{
				System.out.println(mergeStatus);
				mergeStatus++;
				if(mergeStatus == 4){
					road.moveVehicle(v, pos, 1);
				}
			}
		}
	}
```

FastDriver

```
public class FastDriver extends DriverLogic {

	public FastDriver(Vehicle v) {
		type="fast";
		v.speed = (int)(.6*v.max_speed);
		maxTimeToImpact = (int)Math.ceil(v.max_speed/v.accel)+1;
		slowTime = 3;
		speedTime = 8;
	}

	public void drive(Road road, Vehicle v) {

		accel(road,v);

		int delta = (int)Math.ceil(mps(v.speed))+1;
		move(delta, road, v);

		mergeLeft(road, v);
	}

	// custom merge-left
	// merge buffer of 4 (fast)
	public void mergeLeft(Road road, Vehicle v){
		if(v.findLane(road) == 1){
			return;
		}
```

```
		int pos = v.findIndex(road);

		if(mergeStatus == 0){
			for(int i=pos+4;i>pos-v.length-4;i--){
				if(i >= 0 && i < road.roadblock_position &&
road.lane1.get(i) == null){
					return;
				}
			}

			mergeStatus++;
		}else{
			System.out.println(mergeStatus);
			mergeStatus++;
			if(mergeStatus == 3){
				road.moveVehicle(v, pos, 1);
			}
		}
	}
}
```

RandomDriver

```
public class RandomDriver extends DriverLogic {

	public RandomDriver(Vehicle v) {
		type="random";
		v.speed = (int)(v.max_speed*.5);

	}

	public void drive(Road road, Vehicle v) {

		int delta = (int)Math.ceil(mps(v.speed))+1;
		move(delta, road, v);

		mergeLeft(road, v);
	}

	// custom merge-left
	// merge buffer of 2 (crashes often)
	public void mergeLeft(Road road, Vehicle v){
```

```
        if(v.findLane(road) == 1){
            return;
        }

        int pos = v.findIndex(road);

        if(mergeStatus == 0){
            for(int i=pos+2;i>pos-v.length-2;i--){
                if(i >= 0 && i < road.roadblock_position &&
road.lane1.get(i) == null){
                    return;
                }
            }

            mergeStatus++;
        }else{
            System.out.println(mergeStatus);
            mergeStatus++;
            if(mergeStatus == 4){
                road.moveVehicle(v, pos, 1);
            }
        }
    }
}
```

RoadBlock

```
public class RoadBlock extends Vehicle {

    public RoadBlock() {
        logic = new DriverLogic();
        exited = true;
        speed = 0;
    }

    public void drive(){

    }

}
```

SafeDriver

```
public class SafeDriver extends DriverLogic {

	public SafeDriver(Vehicle v) {
		type="Safe";
		v.speed = (int)(.4*v.max_speed);	//v.max_speed);
		maxTimeToImpact = (int)Math.ceil(v.max_speed/v.accel)+4;
		slowTime = 4;
		speedTime = 10;

	}

	public void drive(Road road, Vehicle v) {
		accel(road,v);

		int delta = (int)Math.ceil(mps(v.speed))+1;
		move(delta, road, v);

		mergeLeft(road, v);
	}

	// custom merge-left
	// merge buffer of 10 (most safe)
	public void mergeLeft(Road road, Vehicle v){
		if(v.findLane(road) == 1){
			return;
		}

		int pos = v.findIndex(road);

		if(mergeStatus == 0){
			for(int i=pos+10;i>pos-v.length-10;i--){
				if(i >= 0 && i < road.roadblock_position &&
road.lane1.get(i) == null){
					return;
				}
			}

			mergeStatus++;
		}else{
			System.out.println(mergeStatus);
			mergeStatus++;
```

```
                if(mergeStatus == 4){
                     road.moveVehicle(v, pos, 1);
                }
           }
      }
}
```

Sedan

```
public class Sedan extends Vehicle{


      public Sedan() {
           length = 3;
           max_speed = 100;
           brakeAccel = -9.5;
      }


}
```

SelfishDriver

```
public class SelfishDriver extends DriverLogic{

      public SelfishDriver(Vehicle v) {
           type="selfish";
           v.speed = (int)(.6*v.max_speed);     //v.max_speed);
           maxTimeToImpact = (int)Math.ceil(v.max_speed/v.accel)+3;
           slowTime = 4;
           speedTime = 9;
      }

      public void drive(Road road, Vehicle v) {

           accel(road,v);

           int delta = (int)Math.ceil(mps(v.speed))+1;
           move(delta, road, v);
```

```
            mergeLeft(road, v);
      }

      // custom merge-left
      // merge buffer of 3 (only cares about himself)
      public void mergeLeft(Road road, Vehicle v){
            if(v.findLane(road) == 1){
                  return;
            }

            int pos = v.findIndex(road);

            if(mergeStatus == 0){
                  for(int i=pos+3;i>pos-v.length-3;i--){
                        if(i >= 0 && i < road.roadblock_position &&
road.lane1.get(i) == null){
                              return;
                        }
                  }

                  mergeStatus++;
            }else{
                  System.out.println(mergeStatus);
                  mergeStatus++;
                  if(mergeStatus == 4){
                        road.moveVehicle(v, pos, 1);
                  }
            }
      }
}
```

SlowDriver

```
public class SlowDriver extends DriverLogic {

      public SlowDriver(Vehicle v) {
            type="slow";
            v.speed = (int)(.2*v.max_speed);      //v.max_speed);
            maxTimeToImpact = (int)Math.ceil(v.max_speed/v.accel)+6;
            slowTime = 8;
            speedTime = 16;
      }
```

```
	public void drive(Road road, Vehicle v) {

		accel(road,v);

		int delta = (int)Math.ceil(mps(v.speed))+1;
		move(delta, road, v);

		mergeLeft(road, v);
	}

	// custom merge-left
	// merge buffer of 8 (takes time to merge)
	public void mergeLeft(Road road, Vehicle v){
		if(v.findLane(road) == 1){
			return;
		}

		int pos = v.findIndex(road);

		if(mergeStatus == 0){
			for(int i=pos+8;i>pos-v.length-8;i--){
				if(i >= 0 && i < road.roadblock_position &&
road.lane1.get(i) == null){
					return;
				}
			}

			mergeStatus++;
		}else{
			System.out.println(mergeStatus);
			mergeStatus++;
			if(mergeStatus == 7){
				road.moveVehicle(v, pos, 1);
			}
		}
	}
}
```

TestDriver

```
public class TestDriver extends DriverLogic {

	public TestDriver(Vehicle v) {
		type="test";
		v.speed = (int)(.7*v.max_speed);

	}

	public void drive(Road road, Vehicle v) {

		double congT = 0;	// get congestion value from road
		double cong1 = 0;	// congestion in front
		double cong2 = 0;	// congestion in front in other lane
		double cong3 = 0;	// congestion in back in other lane
		double cong4 = 0;	// congestion in back

		double vel1 = 0;	// congestion in front
		double vel2 = 0;	// congestion in front in other lane
		double vel3 = 0;	// congestion in back in other lane
		double vel4 = 0;	// congestion in back

		int cars = road.vehicles.size();

		for (Vehicle c: road.vehicles) {
			if (road.vehicles.indexOf(v) > 
road.vehicles.indexOf(c)) {	// current vehicle is behind the test 
vehicle
				if (v.findLane(road) == c.findLane(road)) {
					vel4+=c.speed;
				} else {
					vel3+=c.speed;
				}
			} else if (road.vehicles.indexOf(v) < 
road.vehicles.indexOf(c)) {	// current vehicle is in front of the 
test vehicle
				if (v.findLane(road) == c.findLane(road)) {
					vel1+=c.speed;
				} else {
					vel2+=c.speed;
				}
			}
		}
```

```
        vel1 /= cars;
        vel2 /= cars;
        vel3 /= cars;
        vel4 /= cars;

        int frontLen = 3000 - v.findIndex(road);
        int backLen = v.findIndex(road);

        cong1 = congestion(vel1,cars,frontLen,1);
        cong2 = congestion(vel2,cars,frontLen,1);
        cong3 = congestion(vel3,cars,backLen,1);
        cong4 = congestion(vel4,cars,backLen,1);

        // all congestion values have been computed and stored at
this point

        double currCongDiff = congDiff(congT, cong1, cong2, cong3,
cong4);
        double congDiff1 = congDiff(congT,
congestion((vel1*cars+v.speed)/cars,cars,frontLen-v.speed,1), cong2,
cong3, cong4);
        double congDiff2 = congDiff(congT, cong1,
congestion((vel2*cars+v.speed)/cars,cars,frontLen-v.speed,1), cong3,
cong4);
        double congDiff3 = congDiff(congT, cong1, cong2,
congestion((vel3*cars+v.speed)/cars,cars,backLen+v.speed,1), cong4);
        double congDiff4 = congDiff(congT, cong1, cong2, cong3,
congestion((vel4*cars+v.speed)/cars,cars,backLen+v.speed, 1));

        double minValue =
Math.min(Math.min(congDiff1,congDiff2),Math.min(Math.min(congDiff3,co
ngDiff4),currCongDiff));

        if (minValue == currCongDiff) {
            actions.add("stay");
        } else if (minValue == congDiff1) {
            actions.add("accel");
            v.speed+=v.accel;
        } else if (minValue == congDiff2) {
            if (v.findLane(road) == 1) {
                actions.add("attempt move right");
                mergeRight(road, v);
            } else {
```

```
				mergeLeft(road, v);
			}
		} else if (minValue == congDiff3) {
			actions.add("move back and merge");
			v.speed+=v.brakeAccel;
			if (v.findLane(road) == 1) {
				mergeRight(road, v);
			} else {
				mergeLeft(road, v);
			}
		} else if (minValue == congDiff4) {
			actions.add("brake");
			v.speed+=v.brakeAccel;
		}

		int delta = (int)Math.ceil(mps(v.speed))+1;
		move(delta, road, v);
	}

	public double congestion(double vel, int cars, int len, int
lanes) {
		return vel*vel*cars/len/lanes/2;
	}

	public double congDiff(double ct, double c1, double c2, double
c3, double c4) {
		return Math.abs(ct-c1) + Math.abs(ct-c2) + Math.abs(ct-c3)
+ Math.abs(ct-c4);
	}
}
```

Truck

```
public class Truck extends Vehicle{


	public Truck() {
		length = 10;
		max_speed = 65;
		brakeAccel = -6.5;
	}
}
```

Van

```
public class Van extends Vehicle{

	public Van() {
		length = 4;
		max_speed = 80;
		brakeAccel = -7.2;
	}

}
```

Model 2:

Driver

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileOutputStream;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;

public class Driver {
	private static PrintWriter file; // for sets of tests
	private static FileWriter masterFile; // for overall analysis
of data
	private static BufferedReader reader;

	private static int num_tests = 5;

	public static int num_cars = 0;
	private static int num_sedans = 10;
	private static int num_vans = 0;
	private static int num_trucks = 0;

	// dont change these values
	private static int num_random_drivers = 0;
```

```
	private static int num_aggressive_drivers = 0;
	private static int num_conservative_drivers = 0;
	private static int num_slow_drivers = 0;
	private static int num_fast_drivers = 0;
	private static int num_safe_drivers = 0;
	private static int num_selfish_drivers = 0;
	private static int num_fair_drivers = 0;
	private static int num_test_drivers = 0;

	private static String crash_result = "";

	public static void main(String[] args) throws IOException {
		// each set of tests gets printed out in file

		String text = "";
		try {
			reader = new BufferedReader(new
FileReader("master_file.csv"));
			try {
			    StringBuilder builder = new StringBuilder();
			    String line = reader.readLine();

			    while (line != null) {
			        builder.append(line);
			        builder.append(System.lineSeparator());
			        line = reader.readLine();
			    }
			    text = builder.toString();
			} finally {
			    reader.close();
			}
		} catch(Exception e) {System.out.println("master file read
error");}

		String temp = "";
		file = null;
		for (int i=1111; i<9999; i++) {
			if (!text.contains(""+i)) {
				temp = ""+i+".csv";
				break;
			}
		}
		try {
```

```
                file = new PrintWriter(temp);
            } catch (Exception e) {System.out.println("file writing
error");}
            try {
                masterFile = new FileWriter("master_file.csv", true);
            } catch (Exception e) {System.out.println("master file
write error");}

            // masterFile.append("TEST FILE,# CARS,# SEDANS,# VANS,#
TRUCKS,# RANDOM DRIVERS,# AGGRESSIVE DRIVERS,# CONSERVATIVE DRIVERS,#
SLOW DRIVERS,# FAST DRIVERS,# SAFE DRIVERS,# SELFISH DRIVERS,# FAIR
DRIVERS,# TEST DRIVERS,AVG. TOTAL RUNTIME,AVG. AVG. TIME/CAR,#
CRASHES,AVG AVG ANGER\n");

            /* SET UP TESTS */

            num_cars = num_sedans + num_vans + num_trucks;
            int total_runtime = 0;
            double avg_time_per_car = 0;

            // beginning of test
            file.println("TOTAL RUNTIME,AVG TIME PER CAR,CRASHED,AVG
ANGER");

            num_cars = num_sedans + num_vans + num_trucks;

            double avg_runtime = 0.0;
            double avg_avg_time_per_car = 0.0;
            int num_crashes = 0;
            double avg_avg_anger = 0;

            for (int i=0; i<num_tests; i++) {

                num_random_drivers = 0;
                num_aggressive_drivers = 0;
                num_conservative_drivers = 0;
                num_slow_drivers = 0;
                num_fast_drivers = 0;
                num_safe_drivers = 0;
                num_selfish_drivers = 0;
                num_fair_drivers = 0;
                num_test_drivers = 0;
```

```
		Road r = new Road();

		//r.createVehicle("sedan", "test", 5, 2);

		int pos;
		int lane;
		for(int j=0;j<num_sedans;j++){
			pos = (int)(Math.random()*1500)+15;
			lane = (int)(Math.random()*3)+1;
			System.out.println(pos + " " + lane);
			int driverType = (int)(Math.random()*7)+1;
			switch(driverType) {
			case 1:
				if (!r.createVehicle("sedan",
"aggressive", pos, lane)) {
					j--;
				}
				num_aggressive_drivers++;
				System.out.println("created aggressive");
				break;
			case 2:
				if (!r.createVehicle("sedan",
"conservative", pos, lane)) {
					j--;
				}
				num_conservative_drivers++;
				System.out.println("created
conservative");
				break;
			case 3:
				if (!r.createVehicle("sedan", "fair", pos,
lane)) {
					j--;
				}
				num_fair_drivers++;
				System.out.println("created fair");
				break;
			case 4:
				if (!r.createVehicle("sedan", "fast", pos,
lane)) {
					j--;
				}
				num_fast_drivers++;
```

```
                    System.out.println("created fast");
                    break;
               case 5:
                    if (!r.createVehicle("sedan", "safe", pos,
lane)) {
                         j--;
                    }
                    num_safe_drivers++;
                    System.out.println("created safe");
                    break;
               case 6:
                    if (!r.createVehicle("sedan", "selfish",
pos, lane)) {
                         j--;
                    }
                    num_selfish_drivers++;
                    System.out.println("created selfish");
                    break;
               case 7:
                    if (!r.createVehicle("sedan", "slow", pos,
lane)) {
                         j--;
                    }
                    num_slow_drivers++;
                    System.out.println("created slow");
                    break;
               default:
                    if (!r.createVehicle("sedan", "random",
pos, lane)) {
                         j--;
                    }
                    num_random_drivers++;
                    break;
               }
          }

          for(int j=0;j<num_vans;j++){
               pos = (int)(Math.random()*1500)+15;
               lane = (int)(Math.random()*3)+1;
               System.out.println(pos + " " + lane);
               int driverType = (int)(Math.random()*7)+1;
               switch(driverType) {
               case 1:
```

```
                                        if (!r.createVehicle("van", "aggressive",
pos, lane)) {
                                                j--;
                                        }
                                        num_aggressive_drivers++;
                                        break;
                                case 2:
                                        if (!r.createVehicle("van",
"conservative", pos, lane)) {
                                                j--;
                                        }
                                        num_conservative_drivers++;
                                        System.out.println("created
conservative");
                                        break;
                                case 3:
                                        if (!r.createVehicle("van", "fair", pos,
lane)) {
                                                j--;
                                        }
                                        num_fair_drivers++;
                                        System.out.println("created fair");
                                        break;
                                case 4:
                                        if (!r.createVehicle("van", "fast", pos,
lane)) {
                                                j--;
                                        }
                                        num_fast_drivers++;
                                        System.out.println("created fast");
                                        break;
                                case 5:
                                        if (!r.createVehicle("van", "safe", pos,
lane)) {
                                                j--;
                                        }
                                        num_safe_drivers++;
                                        System.out.println("created safe");
                                        break;
                                case 6:
                                        if (!r.createVehicle("van", "selfish",
pos, lane)) {
                                                j--;
```

```
                    }
                    num_selfish_drivers++;
                    System.out.println("created selfish");
                    break;
               case 7:
                    if (!r.createVehicle("van", "slow", pos,
lane)) {
                         j--;
                    }
                    num_slow_drivers++;
                    System.out.println("created slow");
                    break;
               default:
                    if (!r.createVehicle("vans", "random",
pos, lane)) {
                         j--;
                    }
                    num_random_drivers++;
                    break;
               }
          }

          for(int j=0;j<num_trucks;j++){
               pos = (int)(Math.random()*1500)+15;
               lane = (int)(Math.random()*3)+1;
               System.out.println(pos + " " + lane);
               int driverType = (int)(Math.random()*7)+1;
               switch(driverType) {
               case 1:
                    if (!r.createVehicle("truck",
"aggressive", pos, lane)) {
                         j--;
                    }
                    num_aggressive_drivers++;
                    break;
               case 2:
                    if (!r.createVehicle("truck",
"conservative", pos, lane)) {
                         j--;
                    }
                    num_conservative_drivers++;
                    System.out.println("created
conservative");
```

```
                    break;
               case 3:
                    if (!r.createVehicle("truck", "fair", pos,
lane)) {
                         j--;
                    }
                    num_fair_drivers++;
                    System.out.println("created fair");
                    break;
               case 4:
                    if (!r.createVehicle("truck", "fast", pos,
lane)) {
                         j--;
                    }
                    num_fast_drivers++;
                    System.out.println("created fast");
                    break;
               case 5:
                    if (!r.createVehicle("truck", "safe", pos,
lane)) {
                         j--;
                    }
                    num_safe_drivers++;
                    System.out.println("created safe");
                    break;
               case 6:
                    if (!r.createVehicle("truck", "selfish",
pos, lane)) {
                         j--;
                    }
                    num_selfish_drivers++;
                    System.out.println("created selfish");
                    break;
               case 7:
                    if (!r.createVehicle("truck", "slow", pos,
lane)) {
                         j--;
                    }
                    num_slow_drivers++;
                    System.out.println("created slow");
                    break;
               default:
```

```
                    if (!r.createVehicle("trucks", "random",
pos, lane)) {
                        j--;
                    }
                    num_random_drivers++;
                    break;
                }
            }

            // place roadblocks
            r.createVehicle("roadblock", "",
r.roadblock_position, 1);
            r.createVehicle("roadblock", "",
r.roadblock_position, 2); // add this for 3 -> 1

            // create test vehicle
            r.createVehicle("truck", "test", 0, 3);
            // r.createVehicle("sedan", "test", 1000, 3);

            r.orderVehicles();

            r.run();

            double avg_anger = 0;
            for(Vehicle v: r.vehicles){
                avg_anger += v.anger;
            }
            avg_anger = avg_anger/r.vehicles.size();
            avg_avg_anger += avg_anger;

            total_runtime = r.total_runtime;
            avg_runtime += total_runtime;

            String crash_result = (r.crashed) ?
r.crash_vehicle.logic.toString() : ""; // TYPE OF DRIVER THAT CRASHES
            if(!r.crashed){
                avg_time_per_car = r.avg_time_per_car;
                avg_avg_time_per_car += avg_time_per_car;
            }else{
                num_crashes++;
            }
```

```
                file.println(total_runtime + "," + avg_time_per_car +
"," + crash_result + "," + avg_anger);
            }

            avg_runtime = ((double) avg_runtime)/num_tests;
            avg_avg_time_per_car = ((double)
avg_avg_time_per_car)/num_tests;
            avg_avg_anger = ((double) avg_avg_anger)/num_tests;
            System.out.println("--Overall Statistics--");
            System.out.println("Average Runtime:" + avg_runtime);
            System.out.println("Average Average Time/Car: " +
avg_avg_time_per_car);


            System.out.println("Total Crashes: " + num_crashes);
            masterFile.append(temp + "," + num_cars + "," + num_sedans
+ "," + num_vans + "," + num_trucks + "," + num_random_drivers + ","
+ num_aggressive_drivers + "," + num_conservative_drivers + "," +
num_slow_drivers + "," + num_fast_drivers + "," + num_safe_drivers +
"," + num_selfish_drivers + "," + num_fair_drivers + "," +
num_test_drivers + "," + avg_runtime + "," + avg_avg_time_per_car +
"," + num_crashes + "," + avg_avg_anger + "\n");

            // end of test

            masterFile.close();
            file.close();
      }

}
```

Road

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

// 2 constructors
// arguments: num of each type of car, list of vehicles

public class Road {
     public int num_cars; // number of cars
```

```
	public int speed_limit = 65; // speed limit of road
	public int road_length = 3000; // road length - in meters
	public int num_lanes = 3; // number of lanes
	public int roadblock_position = 2500; // position of road-block
from beginning of road

	public Vehicle crash_vehicle; // vehicle that caused crash

	// add up to num_cars
	public int num_sedans = 0;
	public int num_vans = 0;
	public int num_trucks = 0;

	// lanes
	public List<Vehicle> lane1 = new ArrayList<Vehicle>();
	public List<Vehicle> lane2 = new ArrayList<Vehicle>();
	public List<Vehicle> lane3 = new ArrayList<Vehicle>(); // has
roadblock

	// vehicle array
	public List<Vehicle> vehicles = new ArrayList<Vehicle>(); //
adds all vehicles created

	// universal time for simulation
	public int time = 0;
	public List<Double> total_congestions = new
ArrayList<Double>();

	// return information
	public int total_runtime;
	public double avg_time_per_car;

	public boolean crashed = false;

	public Road(int vehicles){
		initLanes();
		for (int i=0; i<vehicles; i++) {
			int vType = (int)Math.random()*3+1;
			String type = "";
			switch(vType){
			case 1:
				type = "sedan";
				break;
```

```
			case 2:
				type = "van";
				break;
			case 3:
				type = "truck";
				break;
			}
			int vDriver = (int)Math.random()*9+1;
			String driverType = "";
			switch(vDriver){
			case 1:
				driverType = "random";
				break;
			case 2:
				driverType = "aggressive";
				break;
			case 3:
				driverType = "conservative";
				break;
			case 4:
				driverType = "slow";
				break;
			case 5:
				driverType = "fast";
				break;
			case 6:
				driverType = "safe";
				break;
			case 7:
				driverType = "selfish";
				break;
			case 8:
				driverType = "fair";
				break;
			case 9:
				driverType = "test";
				break;
			}
			int pos = (int)Math.random()*1500+15;
			int lane = (int)Math.random()*3+1;
			createVehicle(type,driverType,pos,lane);
		}
	}
```

```
	// road constructor for given vehicle arraylist
	public Road(){
		initLanes();
	}

	public boolean createVehicle(String type, String driver, int
pos, int lane) {
		Vehicle newV = null;
		if (type.equals("sedan")) {
			newV = new Sedan();
		}
		else if (type.equals("van")) {
			newV = new Van();
		}
		else if (type.equals("truck")) {
			newV = new Truck();
		}else if(type.equals("roadblock")){
			newV = new RoadBlock();
		}
		if (!placeVehicle(newV, pos, lane)) {
			return false;
		}
		vehicles.add(newV);
		if (driver.equals("random")) {
			newV.logic = new RandomDriver(newV);
		}
		else if (driver.equals("aggressive")) {
			newV.logic = new AggressiveDriver(newV);
		}
		else if (driver.equals("conservative")) {
			newV.logic = new ConservativeDriver(newV);
		}
		else if (driver.equals("slow")) {
			newV.logic = new SlowDriver(newV);
		}
		else if (driver.equals("fast")) {
			newV.logic = new FastDriver(newV);
		}
		else if (driver.equals("safe")) {
			newV.logic = new SafeDriver(newV);
		}
		else if (driver.equals("selfish")) {
```

```
            newV.logic = new SelfishDriver(newV);
        }
        else if (driver.equals("fair")) {
            newV.logic = new FairDriver(newV);
        }
        else if (driver.equals("test")) {
            newV.logic = new TestDriver(newV);
        }
        return true;

    }

    public boolean placeVehicle(Vehicle v, int pos, int lane) {
    // places a vehicle at a specific position (based off of front
index) if the vehicle goes off the array, those elements are left out
        if (lane==1 && lane1.get(pos) == null) {
            for (int i=0; i<v.length; i++) {
                if (pos-i >= 0) {
                    lane1.set(pos-i, v);
                }
            }
            return true;
        }
        else if (lane==2 && lane2.get(pos) == null) {
            for (int i=0; i<v.length; i++) {
                if (pos-i >= 0) {
                    lane2.set(pos-i, v);
                }
            }
            return true;
        }
        else if (lane==3 && lane3.get(pos) == null) {
            for (int i=0; i<v.length; i++) {
                if (pos-i >= 0) {
                    lane3.set(pos-i, v);
                }
            }
            return true;
        }

        return false;
    }
```

```
public void removeVehicle(Vehicle v) {
	while (lane1.indexOf(v) != -1) {
		lane1.set(lane1.indexOf(v), null);
	}
	while (lane2.indexOf(v) != -1) {
		lane2.set(lane2.indexOf(v), null);
	}
	while (lane3.indexOf(v) != -1) {
		lane3.set(lane3.indexOf(v), null);
	}
}

public boolean moveVehicle(Vehicle v, int pos, int lane) {
	removeVehicle(v);
	if (!placeVehicle(v,pos,lane)) {
		crash_vehicle = v;
		v.crashed = true;
		return false;   // crash occurred
	}
	return true;
}

public boolean orderVehicles() {
	// sorts vehicles in descending order
	Collections.sort(vehicles, new CustomComparator());
	return true;
}

public void run() {

	while (nextSecond().equals("running")) {
		time++;
	}

	total_runtime = time;
	System.out.println("Total Runtime: "+time);

	avg_time_per_car = 0;
	for (Vehicle v: vehicles) {
		System.out.println(v.timeOnRoad);
		avg_time_per_car += v.timeOnRoad;
	}
	avg_time_per_car = avg_time_per_car/(vehicles.size());
```

```
		System.out.println("Avg. Time/Car: "+ avg_time_per_car);

		System.out.println("ct size: " +
total_congestions.size());

		// calculate everyone's change in anger
		for(Vehicle v: vehicles){
			System.out.println("here");
			if(!(v instanceof RoadBlock)){
				double delta_a = 0;
				for(int i=0;i<total_congestions.size()-1;i++){
					delta_a +=
(v.subset_congestions.get(i)-total_congestions.get(i));
				}
				v.anger = (1.0/time)*delta_a; // more
congestion, higher anger

				// System.out.println(v.anger);
			}
		}
	}

	// increments through each second
	public String nextSecond(){
		orderVehicles(); // sorts vehicles in descending order

		// System.out.println("time: " + time);

		// congestion calculations
		double avg_velocity = 0;
		for(Vehicle w: vehicles){
			avg_velocity += w.speed;
		}
		avg_velocity = ((double) avg_velocity)/(vehicles.size());
		double ct = congestion(avg_velocity, vehicles.size(),
3000, 3);

		total_congestions.add(ct);

		for (Vehicle v: vehicles) {
			// calculate Cs
			int num_cars_range = 0;
			double avg_velocity_range = 0;
```

```
				// loop range of 100 around car
				for(int
i=v.findIndex(this)-50;i<v.findIndex(this)+50;i++){
					if(i >= 0 && i < 3000){
						if(lane1.get(i) != null){
							num_cars_range++;
							avg_velocity_range +=
lane1.get(i).speed;
						}

						if(lane2.get(i) != null){
							num_cars_range++;
							avg_velocity_range +=
lane2.get(i).speed;
						}

						if(lane3.get(i) != null){
							num_cars_range++;
							avg_velocity_range +=
lane3.get(i).speed;
						}
					}
				}

				if(num_cars_range != 0){
					avg_velocity_range = ((double)
avg_velocity_range)/num_cars_range;
				}else{
					avg_velocity_range = 0;
				}

				double cs = congestion(avg_velocity_range,
num_cars_range, 100, 3);
				v.subset_congestions.add(cs);

				if (v.drive(this)) {  // if a crash took place, then
break out of the simulation
					crashed = true;
					return "crash";
				}
			}
			if (allCarsExited()) {
```

```
                // set anger values
                return "done";
            }
            return "running";          // implies simulation is still
running in this second
      }

      public boolean allCarsExited() {
            for (Vehicle v: vehicles) {
                  if (!v.exited && !v.crashed)
                        return false;
            }
            return true;
      }

      public void initLanes() {
            for (int i=0; i<3000; i++) {
                  lane1.add(null);
                  lane2.add(null);
                  lane3.add(null);
            }
      }

      public double congestion(double vel, int cars, int len, int
lanes) {
            return vel*vel*cars/len/lanes/2;
      }

      private class CustomComparator implements Comparator<Vehicle> {
          @Override
          public int compare(Vehicle v1, Vehicle v2) {
              return
Integer.valueOf(v2.findIndex(Road.this)).compareTo(Integer.valueOf(v1
.findIndex(Road.this))); // descending order
          }
      }
}
```

Vehicle

```
import java.util.ArrayList;
import java.util.List;
```

```
public class Vehicle {
     public double anger = 0;
     public int max_speed;           // the maximum speed attainable
by this kind of car (kph)
     public int speed;                    // the current speed of
this car (kph)
     public int length;                   // the length of this car
     public boolean crashed = false;
     public double brakeAccel;       // the braking acceleration of
this car
     public double accel;            // the acceleration of this car
     public int mergeTime;           // the time (in seconds) in
which this vehicle can merge
     public boolean exited = false;  // whether or not the vehicle
is still in the 3km stretch of road
     public DriverLogic logic;
     public int timeOnRoad = 0;

     public List<Double> subset_congestions = new
ArrayList<Double>();

     public Vehicle() {

     }


     public boolean drive(Road road) {    // returns whether or not
a crash took place
          if (exited) {
               return false;
          }

          if(crashed){
               System.out.println("crashed");
               return true;
          }
          logic.drive(road, this);
          return false;
     }
```

```
	public int findIndex(Road road) {

		if (road.lane1.indexOf(this) != -1) {
			return road.lane1.lastIndexOf(this);
		}
		else if(road.lane2.indexOf(this) != -1){
			return road.lane2.lastIndexOf(this);
		}else{
			return road.lane3.lastIndexOf(this);
		}

	}

	public int findLane(Road road) {
		int lane = 0;
		if (road.lane1.indexOf(this) != -1) {
			lane+=1;
		} if (road.lane2.indexOf(this) != -1) {
			lane+=2;
		}
		if (road.lane3.indexOf(this) != -1) {
			lane+=3;
		}
		return lane;    // returns 1 if car is in lane 1, 2 if
lane 2, or 3 if in both lanes   (0 if vehicle doesn't exist)
	}

}
```

DriverLogic

```
import java.util.ArrayList;
import java.util.List;


public class DriverLogic {

	public String type;

	public int mergeStatus = 0;
	int maxTimeToImpact;
	int slowTime;
	int speedTime;
```

```
	public List<String> actions = new ArrayList<String>();

	public DriverLogic() {

	}

	public void drive(Road road, Vehicle v) {

	}

	public String toString(){
		return "driver logic";
	}

	public int findIndex(Road road, Vehicle v) {

		if (road.lane1.indexOf(v) != -1) {
			return road.lane1.lastIndexOf(v);
		}
		else if(road.lane2.indexOf(v) != -1){
			return road.lane2.lastIndexOf(v);
		}else{
			return road.lane3.lastIndexOf(v);
		}

	}

	public int findLane(Road road, Vehicle v) {
		int lane = 0;
		if (road.lane1.indexOf(v) != -1) {
			lane=1;
		} if (road.lane2.indexOf(v) != -1) {
			lane=2;
		}
		if (road.lane3.indexOf(v) != -1) {
			lane=3;
		}
		return lane;	// returns 1 if car is in lane 1, 2 if
lane 2, or 3 if in both lanes	(0 if vehicle doesn't exist)
	}
```

```
	public boolean move(int delta, Road road, Vehicle v) {
		int newPos = v.findIndex(road);
		if (newPos+delta < 3000) {
			return road.moveVehicle(v, findIndex(road, v)+delta,
findLane(road, v));
		}
		else {
			v.exited=true;
			System.out.println("exited");
			v.timeOnRoad = road.time;
			road.removeVehicle(v);
		}
		return false;
	}

	public double mps(int speed) {  // converts speed (kph) to m/s
		return speed*0.277778;
	}

	public void accel(Road road, Vehicle v) {
		int spacesInFront = 0;
		for (int i=findIndex(road, v)+1; i<3000; i++) {
			if (findLane(road, v) == 1) {
				if (road.lane1.get(i) == null) {
					spacesInFront++;
				} else {
					break;
				}
			}
			else if (findLane(road, v) == 2) {
				if (road.lane2.get(i) == null) {
					spacesInFront++;
				} else {
					break;
				}
			}else if (findLane(road, v) == 3) {
				if (road.lane3.get(i) == null) {
					spacesInFront++;
				} else {
					break;
				}
			}
```

```
		}

		int timeToImpact = (int)(spacesInFront/(v.speed+.01));
		if (maxTimeToImpact - timeToImpact < slowTime) {
			v.speed = v.speed+(int)v.brakeAccel;
			if (v.speed<1) {
				v.speed = 0;
			}
			actions.add("brake");
			// System.out.println("brake");
		} else if (maxTimeToImpact - timeToImpact > speedTime) {
			v.speed = v.speed+(int)v.accel;
			actions.add("accel");
			if (v.speed>v.max_speed) {
				v.speed = v.max_speed;
			}

			if(v.speed>road.speed_limit){
				v.speed = road.speed_limit;
			}
			// System.out.println("accel");
		}
	}

	// default merge-left
	// no merge buffer
	public void mergeLeft(Road road, Vehicle v){
		if(v.findLane(road) == 1){
			return;
		}
		if(v.findLane(road) == 2){
			int pos = v.findIndex(road);

			if(mergeStatus == 0){
				for(int i=pos;i>pos-v.length;i--){
					if(i >= 0 && i < road.roadblock_position
&& road.lane1.get(i) == null){
						return;
					}
				}

				mergeStatus++;
			}else{
```

```
                    mergeStatus++;
                    if(mergeStatus == 4){
                        road.moveVehicle(v, pos, 1);
                    }
                }
            }
            if(v.findLane(road) == 3){
                int pos = v.findIndex(road);

                if(mergeStatus == 0){
                    for(int i=pos;i>pos-v.length;i--){
                        if(i >= 0 && i < road.roadblock_position
&& road.lane2.get(i) == null){
                            return;
                        }
                    }

                    mergeStatus++;
                }else{
                    mergeStatus++;
                    if(mergeStatus == 4){
                        road.moveVehicle(v, pos, 2);
                    }
                }
            }
        }

        public void mergeRight(Road road, Vehicle v){
            if(v.findLane(road) == 3){
                return;
            }

            if(v.findLane(road) == 2){
                int pos = v.findIndex(road);

                if(mergeStatus == 0){
                    for(int i=pos;i>pos-v.length;i--){
                        if(i >= 0 && i < road.roadblock_position -
100 && road.lane3.get(i) == null){
                            return;
                        }
                    }
```

```
                    mergeStatus++;
                }else{
                    mergeStatus++;
                    if(mergeStatus == 4){
                        road.moveVehicle(v, pos, 3);
                    }
                }
            }

            if(v.findLane(road) == 1){
                int pos = v.findIndex(road);

                if(mergeStatus == 0){
                    for(int i=pos;i>pos-v.length;i--){
                        if(i >= 0 && i < road.roadblock_position -
100 && road.lane2.get(i) == null){
                            return;
                        }
                    }

                    mergeStatus++;
                }else{
                    mergeStatus++;
                    if(mergeStatus == 4){
                        road.moveVehicle(v, pos, 2);
                    }
                }
            }
        }

        public double congestion(double vel, int cars, int len, int
lanes) {
            return vel*vel*cars/len/lanes/2;
        }
}
```

AggressiveDriver

```
public class AggressiveDriver extends DriverLogic{
```

```
	public AggressiveDriver(Vehicle v) {
		type="aggressive";
		v.speed = (int)(.6*v.max_speed);
		maxTimeToImpact = (int)Math.ceil(v.max_speed/v.accel)+1;
		slowTime = 3;
		speedTime = 8;
	}


	public void drive(Road road, Vehicle v) {

		accel(road,v);

		int delta = (int)Math.ceil(mps(v.speed))+1;
		move(delta, road, v);

		mergeLeft(road, v);
	}

	// custom merge-left
	// merge buffer of 0
	public void mergeLeft(Road road, Vehicle v){
		if(v.findLane(road) == 1){
			return;
		}
		if(v.findLane(road) == 2){
			int pos = v.findIndex(road);

			if(mergeStatus == 0){
				for(int i=pos;i>pos-v.length;i--){
					if(i >= 0 && i < road.roadblock_position
&& road.lane1.get(i) == null){
						return;
					}
				}

				mergeStatus++;
			}else{
				mergeStatus++;
				if(mergeStatus == 3){
					road.moveVehicle(v, pos, 1);
				}
			}
```

```
		}
		if(v.findLane(road) == 3){
			int pos = v.findIndex(road);

			if(mergeStatus == 0){
				for(int i=pos;i>pos-v.length;i--){
					if(i >= 0 && i < road.roadblock_position
&& road.lane2.get(i) == null){
						return;
					}
				}

				mergeStatus++;
			}else{
				mergeStatus++;
				if(mergeStatus == 3){
					road.moveVehicle(v, pos, 2);
				}
			}
		}
	}
}
```

ConservativeDriver

```
public class ConservativeDriver extends DriverLogic{

	public ConservativeDriver(Vehicle v) {
		type="conservative";
		v.speed = (int)(.4*v.max_speed);	//v.max_speed);
		maxTimeToImpact = (int)Math.ceil(v.max_speed/v.accel)+4;
		slowTime = 6;
		speedTime = 14;
	}

	public void drive(Road road, Vehicle v) {

		accel(road,v);

		int delta = (int)Math.ceil(mps(v.speed))+1;
		move(delta, road, v);
```

```
			mergeLeft(road, v);
		}

		// custom merge-left
		// merge buffer of 5
		public void mergeLeft(Road road, Vehicle v){
			if(v.findLane(road) == 1){
				return;
			}
			if(v.findLane(road) == 2){
				int pos = v.findIndex(road);

				if(mergeStatus == 0){
					for(int i=pos+5;i>pos-v.length-5;i--){
						if(i >= 0 && i < road.roadblock_position
&& road.lane1.get(i) == null){
							return;
						}
					}

					mergeStatus++;
				}else{
					mergeStatus++;
					if(mergeStatus == 4){
						road.moveVehicle(v, pos, 1);
					}
				}
			}
			if(v.findLane(road) == 3){
				int pos = v.findIndex(road);

				if(mergeStatus == 0){
					for(int i=pos+5;i>pos-v.length-5;i--){
						if(i >= 0 && i < road.roadblock_position
&& road.lane2.get(i) == null){
							return;
						}
					}

					mergeStatus++;
				}else{
					mergeStatus++;
					if(mergeStatus == 4){
```

```
                        road.moveVehicle(v, pos, 2);
                    }
                }
            }
        }
}
```

FairDriver

```
public class FairDriver extends DriverLogic     {

    public FairDriver(Vehicle v) {
        type="fair";
        v.speed = (int)(.4*v.max_speed);     //v.max_speed);
        maxTimeToImpact = (int)Math.ceil(v.max_speed/v.accel)+2;
        slowTime = 4;
        speedTime = 10;
    }

    public void drive(Road road, Vehicle v) {

        accel(road,v);

        int delta = (int)Math.ceil(mps(v.speed))+1;
        move(delta, road, v);

        mergeLeft(road, v);
    }

    // custom merge-left
    // merge buffer of 5 (reasonable)
    public void mergeLeft(Road road, Vehicle v){
        if(v.findLane(road) == 1){
            return;
        }
        if(v.findLane(road) == 2){
            int pos = v.findIndex(road);

            if(mergeStatus == 0){
                for(int i=pos+5;i>pos-v.length-5;i--){
                    if(i >= 0 && i < road.roadblock_position
&& road.lane1.get(i) == null){
```

```
                                return;
                            }
                        }

                        mergeStatus++;
                    }else{
                        mergeStatus++;
                        if(mergeStatus == 4){
                            road.moveVehicle(v, pos, 1);
                        }
                    }
                }
                if(v.findLane(road) == 3){
                    int pos = v.findIndex(road);

                    if(mergeStatus == 0){
                        for(int i=pos+5;i>pos-v.length-5;i--){
                            if(i >= 0 && i < road.roadblock_position
&& road.lane2.get(i) == null){
                                return;
                            }
                        }

                        mergeStatus++;
                    }else{
                        mergeStatus++;
                        if(mergeStatus == 4){
                            road.moveVehicle(v, pos, 2);
                        }
                    }
                }
        }
}
```

FastDriver

```
public class FastDriver extends DriverLogic {

    public FastDriver(Vehicle v) {
        type="fast";
        v.speed = (int)(.6*v.max_speed);
        maxTimeToImpact = (int)Math.ceil(v.max_speed/v.accel)+1;
```

```
		slowTime = 3;
		speedTime = 8;
	}

	public void drive(Road road, Vehicle v) {

		accel(road,v);

		int delta = (int)Math.ceil(mps(v.speed))+1;
		move(delta, road, v);

		mergeLeft(road, v);
	}

	// custom merge-left
	// merge buffer of 4 (fast)
	public void mergeLeft(Road road, Vehicle v){
		if(v.findLane(road) == 1){
			return;
		}
		if(v.findLane(road) == 2){
			int pos = v.findIndex(road);

			if(mergeStatus == 0){
				for(int i=pos+4;i>pos-v.length-4;i--){
					if(i >= 0 && i < road.roadblock_position
&& road.lane1.get(i) == null){
						return;
					}
				}

				mergeStatus++;
			}else{
				mergeStatus++;
				if(mergeStatus == 3){
					road.moveVehicle(v, pos, 1);
				}
			}
		}
		if(v.findLane(road) == 3){
			int pos = v.findIndex(road);

			if(mergeStatus == 0){
```

```
                    for(int i=pos+4;i>pos-v.length-4;i--){
                         if(i >= 0 && i < road.roadblock_position
&& road.lane2.get(i) == null){
                              return;
                         }
                    }

                    mergeStatus++;
               }else{
                    mergeStatus++;
                    if(mergeStatus == 3){
                         road.moveVehicle(v, pos, 2);
                    }
               }
          }
     }
}
```

RandomDriver

```
public class RandomDriver extends DriverLogic {

     public RandomDriver(Vehicle v) {
          type="random";
          v.speed = (int)(v.max_speed*.5);

     }

     public void drive(Road road, Vehicle v) {

          int delta = (int)Math.ceil(mps(v.speed))+1;
          move(delta, road, v);

          mergeLeft(road, v);
     }

     // custom merge-left
     // merge buffer of 2 (crashes often)
     public void mergeLeft(Road road, Vehicle v){
          if(v.findLane(road) == 1){
               return;
          }
```

```
		if(v.findLane(road) == 2){
			int pos = v.findIndex(road);

			if(mergeStatus == 0){
				for(int i=pos+2;i>pos-v.length-2;i--){
					if(i >= 0 && i < road.roadblock_position
&& road.lane1.get(i) == null){
						return;
					}
				}

				mergeStatus++;
			}else{
				mergeStatus++;
				if(mergeStatus == 4){
					road.moveVehicle(v, pos, 1);
				}
			}
		}
		if(v.findLane(road) == 3){
			int pos = v.findIndex(road);

			if(mergeStatus == 0){
				for(int i=pos+2;i>pos-v.length-2;i--){
					if(i >= 0 && i < road.roadblock_position
&& road.lane2.get(i) == null){
						return;
					}
				}

				mergeStatus++;
			}else{
				mergeStatus++;
				if(mergeStatus == 4){
					road.moveVehicle(v, pos, 2);
				}
			}
		}
	}
}
```

RoadBlock

```
public class RoadBlock extends Vehicle {

	public RoadBlock() {
		logic = new DriverLogic();
		exited = true;
		speed = 0;
	}

	public void drive(){

	}

}
```

SafeDriver

```
public class SafeDriver extends DriverLogic {

	public SafeDriver(Vehicle v) {
		type="Safe";
		v.speed = (int)(.4*v.max_speed);	//v.max_speed);
		maxTimeToImpact = (int)Math.ceil(v.max_speed/v.accel)+4;
		slowTime = 4;
		speedTime = 10;

	}

	public void drive(Road road, Vehicle v) {
		accel(road,v);

		int delta = (int)Math.ceil(mps(v.speed))+1;
		move(delta, road, v);

		mergeLeft(road, v);
	}

	// custom merge-left
	// merge buffer of 10 (most safe)
	public void mergeLeft(Road road, Vehicle v){
		if(v.findLane(road) == 1){
			return;
```

```
			}
			if(v.findLane(road) == 2){
				int pos = v.findIndex(road);

				if(mergeStatus == 0){
					for(int i=pos+10;i>pos-v.length-10;i--){
						if(i >= 0 && i < road.roadblock_position
&& road.lane1.get(i) == null){
							return;
						}
					}

					mergeStatus++;
				}else{
					mergeStatus++;
					if(mergeStatus == 4){
						road.moveVehicle(v, pos, 1);
					}
				}
			}
			if(v.findLane(road) == 3){
				int pos = v.findIndex(road);

				if(mergeStatus == 0){
					for(int i=pos+10;i>pos-v.length-10;i--){
						if(i >= 0 && i < road.roadblock_position
&& road.lane2.get(i) == null){
							return;
						}
					}

					mergeStatus++;
				}else{
					mergeStatus++;
					if(mergeStatus == 4){
						road.moveVehicle(v, pos, 2);
					}
				}
			}
		}
}
```

Sedan

```
public class Sedan extends Vehicle{


      public Sedan() {
            length = 3;
            max_speed = 100;
            brakeAccel = -9.5;
      }


}
```

SelfishDriver

```
public class SelfishDriver extends DriverLogic{

      public SelfishDriver(Vehicle v) {
            type="selfish";
            v.speed = (int)(.6*v.max_speed);     //v.max_speed);
            maxTimeToImpact = (int)Math.ceil(v.max_speed/v.accel)+3;
            slowTime = 4;
            speedTime = 9;
      }

      public void drive(Road road, Vehicle v) {

            accel(road,v);

            int delta = (int)Math.ceil(mps(v.speed))+1;
            move(delta, road, v);

            mergeLeft(road, v);
      }

      // custom merge-left
      // merge buffer of 3 (only cares about himself)
      public void mergeLeft(Road road, Vehicle v){
            if(v.findLane(road) == 1){
                  return;
            }
```

```
            if(v.findLane(road) == 2){
                  int pos = v.findIndex(road);

                  if(mergeStatus == 0){
                        for(int i=pos+3;i>pos-v.length-3;i--){
                              if(i >= 0 && i < road.roadblock_position
&& road.lane1.get(i) == null){
                                    return;
                              }
                        }

                        mergeStatus++;
                  }else{
                        mergeStatus++;
                        if(mergeStatus == 4){
                              road.moveVehicle(v, pos, 1);
                        }
                  }
            }
            if(v.findLane(road) == 3){
                  int pos = v.findIndex(road);

                  if(mergeStatus == 0){
                        for(int i=pos+3;i>pos-v.length-3;i--){
                              if(i >= 0 && i < road.roadblock_position
&& road.lane2.get(i) == null){
                                    return;
                              }
                        }

                        mergeStatus++;
                  }else{
                        mergeStatus++;
                        if(mergeStatus == 4){
                              road.moveVehicle(v, pos, 2);
                        }
                  }
            }
      }
}
```

SlowDriver

```
public class SlowDriver extends DriverLogic {

	public SlowDriver(Vehicle v) {
		type="slow";
		v.speed = (int)(.2*v.max_speed);	//v.max_speed);
		maxTimeToImpact = (int)Math.ceil(v.max_speed/v.accel)+6;
		slowTime = 8;
		speedTime = 16;
	}

	public void drive(Road road, Vehicle v) {

		accel(road,v);

		int delta = (int)Math.ceil(mps(v.speed))+1;
		move(delta, road, v);

		mergeLeft(road, v);
	}

	// custom merge-left
	// merge buffer of 8 (takes time to merge)
	public void mergeLeft(Road road, Vehicle v){
		if(v.findLane(road) == 1){
			return;
		}
		if(v.findLane(road) == 2){
			int pos = v.findIndex(road);

			if(mergeStatus == 0){
				for(int i=pos+8;i>pos-v.length-8;i--){
					if(i >= 0 && i < road.roadblock_position
&& road.lane1.get(i) == null){
						return;
					}
				}

				mergeStatus++;
			}else{
				mergeStatus++;
				if(mergeStatus == 7){
					road.moveVehicle(v, pos, 1);
```

```
                    }
                }
            }
            if(v.findLane(road) == 3){
                int pos = v.findIndex(road);

                if(mergeStatus == 0){
                    for(int i=pos+8;i>pos-v.length-8;i--){
                        if(i >= 0 && i < road.roadblock_position
&& road.lane2.get(i) == null){
                            return;
                        }
                    }

                    mergeStatus++;
                }else{
                    mergeStatus++;
                    if(mergeStatus == 7){
                        road.moveVehicle(v, pos, 2);
                    }
                }
            }
        }
}
```

TestDriver

```
public class TestDriver extends DriverLogic {

    public TestDriver(Vehicle v) {
        type="test";
        v.speed = (int)(.7*v.max_speed);

    }

    public void drive(Road road, Vehicle v) {

        double congT = 0;    // get congestion value from road
        double cong1 = 0;    // congestion in front
        double cong2 = 0;    // congestion in front in other lane
        double cong3 = 0;    // congestion in back in other lane
        double cong4 = 0;    // congestion in back
```

```
        double vel1 = 0;      // congestion in front
        double vel2 = 0;      // congestion in front in other lane
        double vel3 = 0;      // congestion in back in other lane
        double vel4 = 0;      // congestion in back

        int cars = road.vehicles.size();

        for (Vehicle c: road.vehicles) {
             if (road.vehicles.indexOf(v) >
road.vehicles.indexOf(c)) {      // current vehicle is behind the test
vehicle
                  if (v.findLane(road) == c.findLane(road)) {
                       vel4+=c.speed;
                  } else {
                       vel3+=c.speed;
                  }
             } else if (road.vehicles.indexOf(v) <
road.vehicles.indexOf(c)) {      // current vehicle is in front of the
test vehicle
                  if (v.findLane(road) == c.findLane(road)) {
                       vel1+=c.speed;
                  } else {
                       vel2+=c.speed;
                  }
             }
        }

        vel1 /= cars;
        vel2 /= cars;
        vel3 /= cars;
        vel4 /= cars;

        int frontLen = 3000 - v.findIndex(road);
        int backLen = v.findIndex(road);

        cong1 = congestion(vel1,cars,frontLen,3);
        cong2 = congestion(vel2,cars,frontLen,3);
        cong3 = congestion(vel3,cars,backLen,3);
        cong4 = congestion(vel4,cars,backLen,3);

        // all congestion values have been computed and stored at
this point
```

```
            double currCongDiff = congDiff(congT, cong1, cong2, cong3,
cong4);
            double congDiff1 = congDiff(congT,
congestion((vel1*cars+v.speed)/cars,cars,frontLen-v.speed,1), cong2,
cong3, cong4);
            double congDiff2 = congDiff(congT, cong1,
congestion((vel2*cars+v.speed)/cars,cars,frontLen-v.speed,1), cong3,
cong4);
            double congDiff3 = congDiff(congT, cong1, cong2,
congestion((vel3*cars+v.speed)/cars,cars,backLen+v.speed,1), cong4);
            double congDiff4 = congDiff(congT, cong1, cong2, cong3,
congestion((vel4*cars+v.speed)/cars,cars,backLen+v.speed, 1));

            double minValue =
Math.min(Math.min(congDiff1,congDiff2),Math.min(Math.min(congDiff3,co
ngDiff4),currCongDiff));

            if (minValue == currCongDiff) {
                  actions.add("stay");
            } else if (minValue == congDiff1) {
                  actions.add("accel");
                  v.speed+=v.accel;
            } else if (minValue == congDiff2) {
                  if (v.findLane(road) == 1) {
                        actions.add("attempt move right");
                        mergeRight(road, v);
                  } else {
                        mergeLeft(road, v);
                  }
            } else if (minValue == congDiff3) {
                  actions.add("move back and merge");
                  v.speed+=v.brakeAccel;
                  if (v.findLane(road) == 1) {
                        mergeRight(road, v);
                  } else {
                        mergeLeft(road, v);
                  }
            } else if (minValue == congDiff4) {
                  actions.add("brake");
                  v.speed+=v.brakeAccel;
            }
```

```
		int delta = (int)Math.ceil(mps(v.speed))+1;
		move(delta, road, v);
	}

	public double congestion(double vel, int cars, int len, int 
lanes) {
		return vel*vel*cars/len/lanes/2;
	}

	public double congDiff(double ct, double c1, double c2, double 
c3, double c4) {
		return Math.abs(ct-c1) + Math.abs(ct-c2) + Math.abs(ct-c3) 
+ Math.abs(ct-c4);
	}
}
```

Truck

```
public class Truck extends Vehicle{


	public Truck() {
		length = 10;
		max_speed = 65;
		brakeAccel = -6.5;
	}
}
```

Van

```
public class Van extends Vehicle{

	public Van() {
		length = 4;
		max_speed = 80;
		brakeAccel = -7.2;
	}

}
```