# 1 A Letter to the Governor of the State of Washington

Dear Sir/Madam,

We have heard that the governor of the state of Washington are asking of analysis of the effects of allowing self-driving, cooperating cars on the roads listed above in Thurston, Pierce, King, and Snohomish counties. After several days study and consideration, we are glad to submit our solution and we hope that will be good to the traffic condition in United State. Our team has designed a complete model and worked out a series of detailed analysis. We have taken account of various factors that may have effects on the traffic flow including the number of lanes, the peak or average hours and different percentage of vehicles using self-driving, cooperating systems. We also propose policy that will improve traffic and enlarge traffic flow. Our goal is to provide an elaborate plan to include self-driving, cooperating cars in the present traffic and alleviate the traffic pressure under the circumstances that the number of lanes or roads does not change. In this letter, we will introduce a brief view of our modeling approach and the final result to you.

Here is the modeling approach:

1. The simulation model is based on the principle of the Cellular Automaton.
2. We first simulate the initial distribution on the whole Interstates 5, 90, and 405, as well as State Route 520. This initial condition is built on the average daily traffic counts in 2015.
3. We set reasonable rules to control the behavior of the non-self-driving cars on the highway and record its travel parameters.
4. Cars with self-driving, cooperating system are added by 10% to 90% with 20% interval.

The analysis way is as follows:

1. The spatiotemporal pattern is plotted for different percentage of self-driving, cooperating cars to give an approximate view of the whole process.
2. The histogram is plotted to show the difference between the regular hours and the peak hours.
3. The parameters of the individual cars are calculated into three main factors reflecting the traffic condition: traffic flow, average velocity and traffic density.
4. The three main factors are analyzed by Fuzzy Synthetic Evaluation and Comprehensive Evaluation Index is worked out.
5. The line chart of Comprehensive Evaluation Index for different percentage of self-driving, cooperating cars shows an elaborate answer to looking for an equilibria and tipping point.

Our result shows that the increasing of self-driving, cooperating cars exactly eases the traffic pressure. Under some conditions, dedicating a lane to self-driving, cooperating cars is helpful.

Hope our solution can alleviate the traffic pressure and make the highway in the United State flows freely.

Yours faithfully,
Team #58893

# Contents

# 2 Introduction

Traffic capacity is limited in many regions of the United States because the volume of traffic exceeds the designed capacity of the road networks. Autonomous cars are potential to quadruple capacity of highways without increasing number of lanes or roads. This results in less traffic congestion. Besides, autonomous cars can reduce parking space and traffic accidents, benefit the elderly and the disabled and relieve drivers from commuting hours. Thus, its worthy of further analysis.

## 2.1 Restatement of the tasks

We are expected to establish a model simulating traffic flow and to analyze the effect of self-driving cooperating cars. Specifically, we decompose the problem into several sub-problems:

- Construct a model that can simulate traffic flow
- Form a comprehensive evaluation index indicating the performance under each circumstance
- Discuss the influence of the number of lanes
- Discuss the influence of peak and average traffic volume
- Discuss the influence of percentage of vehicles using self-driving, cooperating systems
- Analyze the characteristics of interaction between self-driving and non-self-driving vehicles, including percentage of self-driving vehicles, equilibria, tipping point and exclusively reserved lanes
- Design optimization method to enlarge traffic flow
- Propose policy that will improve traffic

## 2.2 Assumptions and Justifications

To focus on the main problem, we make the following well-justified assumptions.

- **The size of each vehicle is 16 feet $\times$ 6 feet.** Although there are sedan car, van, wagon and SUV, which are of different sizes, we assume that all vehicles on freeways are sedan cars because they are the major components of traffic.
- **All the vehicles uniformly accelerate and decelerate.** It is valueless to consider the process of acceleration and deceleration since it has little effect on the traffic flow. We assume a not-self-driving car has constant acceleration ranging from 5 to 7 $mile/h \cdot s$ while a self-driving car has constant acceleration automatically calculated from the distance from the one immediately ahead. Specifically, all vehicles brake at 81576.6 $mile \cdot h^{-2}$.
- **The distance between two vehicles tends to maintain some specific value.** We assume that all vehicles are spaced compactly, which means they maintain some minimal safety distance and dont spread apart. The detailed calculation method of safety distance is shown in Section 3.2.1. For a not-self-driving vehicle, if the distance from the one immediately ahead is 10% less than the safety distance, the vehicle decelerates and if the real distance is 10% more than the safety distance, the vehicle accelerates. For a self-driving vehicle, as long as the real distance is not the same as safety distance, it adjusts its velocity.
- **We dont consider abnormal phenomena.** Accidents, violation of traffic regulations (e.g. speeding) and any other special circumstance are of negligible researching significance in this case.
- **Either of the two directions accounts for half of the average daily traffic.** We assume same traffic volume for both sides of the roads.
- **Peak traffic hour lasts for approximately an hour.** According to our experiences in daily life, peak traffic hour usually lasts for an hour. Besides, we know that 8% of the daily traffic volume occurs during peak travel hours on average. If peak traffic hour lasts for two hours or even longer, then 4.18% or more of the traffic volume occurs during regular time, which is higher than that of peak hours.

- **Not-self-driving cars decelerate randomly.** Without randomization, the model is a deterministic algorithm, i.e., the cars always move in a set pattern once the original state of the road is set, which is not a real-world case. Besides, with randomization, this model can simulate otherwise absent phantom traffic jams [1]. We set the possibility of deceleration as 0.3.
- **Response time for human drivers is approximately 1s.** Humans, unlike self-driving cars, need time to react. Usually, it takes 0.75-1.5s for a driver to take action, so we set the response time as 1s.
- **All cars are exactly the calculated safety distance away from the one immediately ahead.** When the model is constructed, we need to input the initial state of the road. We assume all cars are initially well-spaced so that we can better observe the evolution of traffic flow under various circumstances.
- **We dont consider lane changing.** The impact of this phenomenon on traffic flow is negligible compared with that of acceleration and deceleration.

### 2.3 Notations

All the variables used in this paper are listed in Table 1.

| Symbol | Definition | Units |
|---|---|---|
| $d$ | the distance between two mileposts, i.e. endMilepost - startMilepost | $mile$ |
| $s$ | stopping distance | $mile$ |
| $l$ | length of a car | $mile$ |
| $v$ | initial velocity of a car | $mile/h$ |
| $c$ | average hourly traffic counts | $1/h$ |
| $a$ | maximum acceleration | $mile/h^2$ |
| $v_{ij,nlane}$ | the average speed of the $j_{th}$ car on the $i_{th}$ section of the freeway which has $nlane$ lanes | $mile/h$ |
| $v_{i,nlane}$ | the average speed of all the vehicles on the $i_{th}$ section of the freeway which has $nlane$ lanes | $mile/h$ |
| $v_{nlane}$ | the average speed of all the vehicles on the freeway which has $nlane$ lanes | $mile/h$ |
| $\rho_{i,nlane}$ | the density of the $i_{th}$ section of the road which has $nlane$ lanes | $1/mile$ |
| $\rho_{nlane}$ | the average density of all the sections of the freeway which has $nlane$ lanes | $1/mile$ |
| $q_{i,lane}$ | the flow of the $lane_{th}$ lane of the $i_{th}$ section of the freeway which has $nlane$ lanes | $1/hour$ |
| $q_{nlane}$ | the average flow of all the sections of the freeway which has $nlane$ lanes | $2/hour$ |
| $nlane$ | the number of lanes | $unitless$ |
| $N_i$ | the total number of vehicles on the $i_{th}$ section of the freeway | $unitless$ |
| $n$ | the number of sections | $unitless$ |
| $n_{i,lane}$ | the number of vehicles on the $lane_{th}$ lane of the $i_{th}$ section of the freeway | $unitless$ |
| $l_i$ | the length of the $i_{th}$ section of the freeway | $mile$ |

Table 1: Symbol Table.

## 3 Model Design and Justification

### 3.1 Simulation Model Based on Cellular Automaton

Cellular automation is always used for studying the moving objects having varying states in a number of discrete time. In this problem we study, the cellular automation is applied in the following aspects:

- The "cell" objects are the travelling cars on the highway.
- Each car has its location that the distance from the startMilepost and the lane it drives on are the coordinates.

[1] Joseph Strombergs explanation of phantom traffic jam is as follows.

*If there are enough cars on a highway, any minor disruptions to the flow of traffic can cause a self-reinforcing chain reaction: one car brakes slightly, and the ones behind it brake just a bit more to avoid hitting it, with the braking eventually amplifying until it produces a wave of stopped or slowed traffic.*[3]

- Each car has a state which is one of moving, accelerating or decelerating.
- Each car's state will be affected by a neighborhood which is the former car in different generations according to some rules.

The simulation model for this problem has similar rules to the cellular automation that

$$\text{vehicle state at time}t = f(\text{former vehicle state at time}(t-1))$$

One of the simplest illustration is shown in Figure 1. Generation 0 shows the initial state of six cells in the grid. Generation 1 shows the state of these six cells after a period of time. In the simulation model shown later more details will be added in such a basic model like the distance between the two cells and other factors concern in the highway traffic.

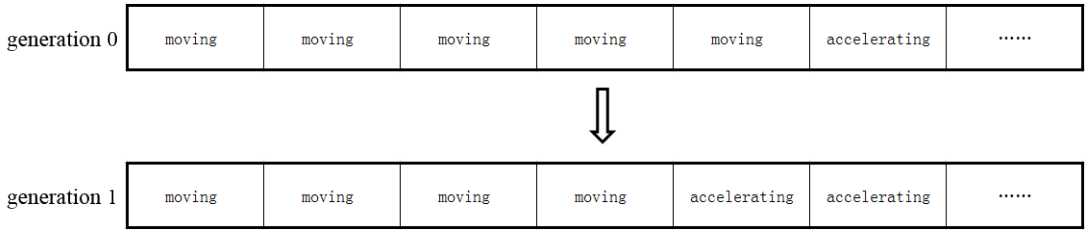


Figure 1: A simple illustration for the state changing dependent on time

## 3.2 Non-Self-Driving Vehicles Model

### 3.2.1 Stopping Distance

In order to keep safe, we assume the distance between any two vehicles is the stopping distance corresponding to the speed of the latter vehicle. When calculate the stopping distance, two factors are considered human factors and vehicle factors.

For the human part, there are two components the human perception time which is the time needed for human to see the hazard and the human reaction time during which the brain convey the signal to the body. The human perception time ranges from 0.25 second to 0.5 second while the human reaction time varies from 0.25 second to 0.75 second.[1] For convenience, we take the sum of these two parts as 1 second.

For the vehicle part, the braking distance concerns the maximum acceleration. We take the braking distance for vehicles at speed 100 kilometer per hour as 32 meters. The maximum acceleration is calculated then:

$$a = \frac{(100\ km \cdot h^{-1})^2}{2 \times 38\ m} = 10.13\ m \cdot s^{-2} = 81576.6\ mile \cdot h^{-2}$$

here we use mile per hour for the convenience of the subsequent calculation.

Then the stopping distance related to the velocity of the vehicle is achieved:

$$s = \frac{v^2}{2a} + \frac{v}{3600}\ mile$$

where $v$ is the velocity of the latter vehicle and $a$ is the maximum acceleration 81576.6 $mile \cdot h^{-2}$.

Note that when the velocity is small enough it is no need to keep the stopping distance above. The main reason is the reaction time will be shortened. We assume the stopping distance is 4 feet as the velocity is below 18.75 mile per hour.

In this model, every vehicle will maintain the exact stopping distance with the former one corresponding to its velocity. The means, while the distance is smaller than the theoretical value, the vehicle will slow down and vice versa. The maximum velocity is limited to 60 mile per hour.

### 3.2.2 Initial Distribution

The initial condition is affected by the traffic counts and the number of lanes.

We first consider the average hourly traffic counts. Since we know the average daily traffic counts (2015) and 8% of the daily traffic volume occurs during peak travel hours. We set the peak travel hour is one hour every day and in this period

$$c = \text{average daily traffic counts} \times 8\%$$

In the other hours, the traffic acounts is

$$c = \text{average daily traffic counts} \times (1 - 8\%)/24 = \text{average daily traffic counts} \times 4\%$$

We know $\frac{d}{s+l} \cdot n$ is the traffic counts between two mileposts in $\frac{d}{v}$ hours. Then we achieve the equation:

$$\frac{d}{s+l} \cdot n \cdot \frac{1}{\frac{d}{v}} = c \implies vn = (s+l)c$$

We have $s = \frac{v^2}{2a} + \frac{v}{3600}\ mile$ and $l = 16\ ft = 0.003\ mile$. After simplification, we have

$$\frac{c}{2a} \cdot v^2 + \left(\frac{c}{3600} - n\right) \cdot v + 0.003 = 0$$

$$s + l = \frac{v^2}{2a} + \frac{v}{3600} + 0.003$$

We use these two equations to determine the initial distribution on the highway. With known average hourly traffic counts and number of the lanes, the initial velocity of a car can be solved, leading to the stopping distance. Initially, there will be $\frac{d}{l+s}$ cars on every lane in this section of the highway and $l + s$ is the distance from the head of the former car and the head of the latter one. All the vehicles on this section of the highway will start to move with the initial speed.

#### 3.2.3 Control Ruleset

After the initial condition is set, we can simulate all the cars under the control rules and record the parameters of every cars including location, acceleration and velocity. These parameters can be used to calculate the average velocity, traffic flow and density of the cars in any section of the highway in future analysis. Here the control ruleset is as follows:

- Every vehicle adjusts the velocity by accelerating or decelerating according to the distance between the former car and itself. When the distance is 10% longer than the stopping distance of its velocity, the vehicle will accelerate. Similarly, it will decelerate when the distance is 10% shorter. 60 mile per hour is the maximum velocity.
- The average daily traffic counts vary among different sections of a highway. This is caused by the difference between the number of the entering cars and the one of the leaving ones. This distinct values of the average daily traffic counts lead to different average velocity in different sections. As the first car of the latter section drives to the last car of the former section, it may have to change its velocity so that it can adapt the traffic condition in the new section. And its change of velocity will affect the state of the following vehicles and leads to a domino effect.
- Since every car is driven by a real human, there should be a randomization deceleration during the process of driving. This will also leads to a domino effect and slow down the cars behind it.

### 3.3 Self-Driving Vehicles Model

#### 3.3.1 Initial Condition

In order to give a clearer comparison with the Non-Self-Driving Vehicles Model, the initial condition doesn't change in this section. This means there will be $\frac{d}{l+s}$ cars on every lane in this section of the highway and $l + s$ is the distance from the head of the former car and the head of the latter one. All the vehicles on this section of the highway will start to move with the initial speed. The distance and the initial speed are calculated by the formula in Non-Self-Driving Vehicles Model.

With the different control ruleset for self-driving vehicles, the simulation will show the real-time traffic situation with different percentage of self-driving cars after a period of time.

#### 3.3.2 Control Ruleset

Due to various sensers and coorperating system on the self-driving vehicles, they can make more precise decisions and the ruleset is different from the human-driving one. Also, self-driving vehicles are more accurate.

- **There exists interaction between selfdriving and non-self-driving vehicles.** The self-driving vehicles don't have a reaction time. When a self-driving vehicle drives behind a non-self-driving vehicle, we suppose the senser can obtain the infromation about the velocity of the former vehicle and the distance between the two vehicles. And to prevent the sudden action of the human-driving vehicle, the vehicle has to keep a stopping distance according to its velocity: $s = \frac{v^2}{2a}$.
- **There exists cooperation between self-driving cars.** When the former vehicle is another self-driving vehicle, the two vehicles can cooperate. Since we are not designing a self-driving car, we omit the detailed velocity changing in the first period of time. We set another standard for the stopping distance and call it safe distance $s_c$. A safe range is assumed around $s_c$. The safe distance and the safe range for different velocity is shown in Table 2.

| velocity[mile/h] | $0 \sim 20$ | $20 \sim 40$ | $40 \sim 60$ |
|---|---|---|---|
| Safe Distance $s_c$[feet] | 4 | 29 | 84 |
| Safe Range[feet] | $0 \sim 4$ | $4 \sim 54$ | more than 49 |

Table 2: The relation between the safe distance, the safe range and velocity

An example about how this works is shown as follows:

For any two self-driving vehicles, if the velocity of the two vehicles both fall on $20 \sim 40$ $mile/h$ and the distance between two cars falls on the corresponding safe range, here it is $4 \sim 54$ $feet$, after a period of time, the distance is set as the safe distance. At first, the latter car will accelerate with the origin acceleration we assume. In the last 0.1 second, it somehow adjust the distance as the safe distance as well as make their velocity the same, which is called a stable situation. In this process, we omit how the self-driving change its acceleration and adjust the distance but focus on the stable situation. As the stable situation is reached, with cooperation, the latter vehicle can just copy the action of the former one until their velocity no longer falls on $20 \sim 40$ $mile/h$, then they change velocity and approach another stable situation. If the distance between the two cars does not fall on the corresponding safe range, the latter vehicle travels with a constant acceleration until the distance enters the safe range.

# 4 Results

## 4.1 Influencing factors

### 4.1.1 Variables and Criteria

There are three variables: number of lanes, peak and/or average traffic volume, and percentage of vehicles using self-driving, cooperating systems. We need a comprehensive evaluation index to indicate the performance on different conditions, so we propose the following criteria:

- **Average speed ($v$)**: the average speed of all vehicles driving on highways with a specific number of lanes over an hour.

$$v_{i,nlane} = \frac{\sum_{j=1}^{N_i} v_{ij,nlane}}{N_i}$$

$$v_{nlane} = \frac{\sum_{i=1}^{n} v_{i,nlane} \times l_i}{\sum_{i=1}^{n} l_i}$$

where $v_{ij,nlane}$ stands for the average speed of the $j_{th}$ car on the $i_{th}$ section of the freeway which has *nlane* lanes, $v_{i,nlane}$ stands for the average speed of all the vehicles on the $i_{th}$ section of the freeway which has *nlane* lanes and $v_{nlane}$ stands for the average speed of all the vehicles on the freeway which has *nlane* lanes. $N_i$ and $n$ stands for the total number of vehicles on the $i_{th}$ section of the freeway and the number of sections. $l_i$ stands for the length of the $i_{th}$ section of the freeway.

- **Density ($\rho$)**: the total number of vehicles per unit length of the freeway.

$$\rho_{i,nlane} = \frac{\sum_{lane=1}^{\#lane} n_{i,lane}}{l_i}$$

$$\rho_{nlane} = \frac{\sum_{i=1}^{n} \rho_{i,nlane}}{n}$$

where $n_{i,lane}$ stands for the number of vehicles on the $lane_{th}$ lane of the $i_{th}$ section of the freeway, $\rho_{i,nlane}$ stands for the density of the $i_{th}$ section of the road which has *nlane* lanes and $\rho_{nlane}$ stands for the average density of all the sections of the freeway which has *nlane* lanes.

- **Flow ($q$)**: the total number of vehicles passing a milepost per hour.

$$q_{i,nlane} = \sum_{lane=1}^{\#lane} q_{i,lane}$$

$$q_{nlane} = \frac{\sum_{i=1}^{n} q_{i,nlane}}{n}$$

where $q_{i,lane}$ stands for the flow of the $lane_{th}$ lane of the $i_{th}$ section of the freeway which has *nlane* lanes and $q_{nlane}$ stands for the average flow of all the sections of the freeway which has *nlane* lanes.

#### 4.1.2 Percentage of the vehicles using self-driving cars and number of lanes

In order to show the effects on the traffic of the percentage of the self-driving, cooperating cars and the number of the lanes, we plot the line charts to show the tendence of the three parameters with the change of these two variables.

In Figure 2, we can have a look at the relation between the velocity and these two variables. Along the x-axis, it shows the different persentage. The self-driving, cooperating cars exactly do good to the traffic condition. We found that the effects of this new kind of cars increase almost linear from 10% to 90% and reaches a maximum at 90%. And it shows a decreasement from 90% to 100%. This figure also shows that two lanes may cause a low average velocity and five landes a high average velocity if other conditions do not change. And the difference is not obvious between three and four lanes.

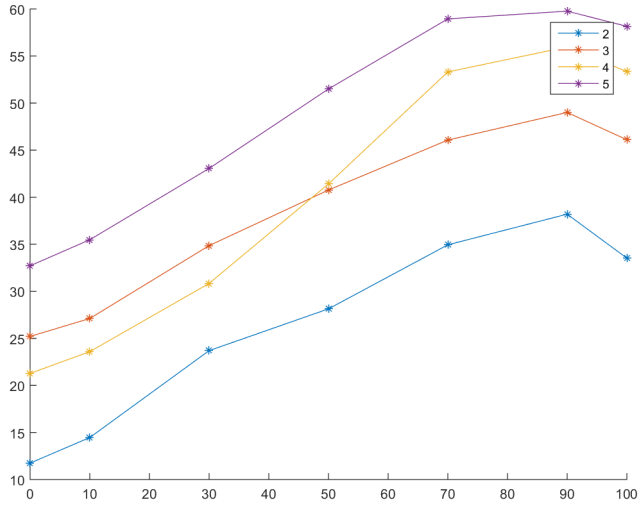


Figure 2: Relation between the velocity and the percentage of the self-driving, cooperating cars and the number of the lanes

In Figure 3, we can have a look at the relation between the flow and these two variables. Again, the self-driving, cooperating cars exactly do good to the traffic condition. We found that the effects of this new kind of cars increase almost linear from 10% to 90%. For lanes 3,4 and 5, there is a tipping point where performance changes markedly from 90% to 100%. However for 2 lanes condition, 90% to 100% contribute little to increase the flow. This figure also shows the more lanes can afford more traffic flow which is understandable.

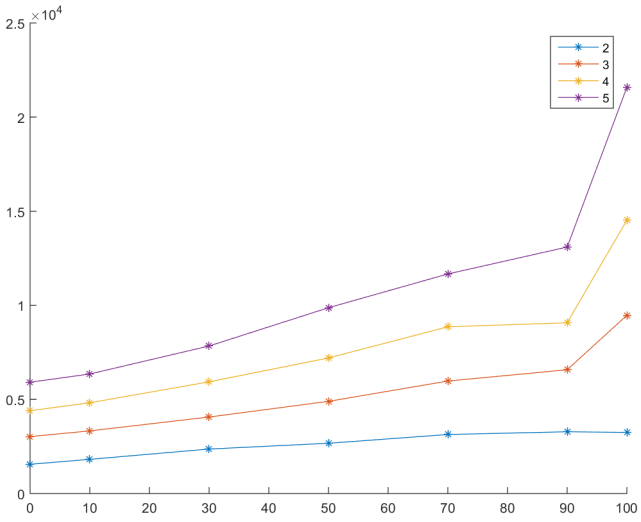


Figure 3: Relation between the flow and the percentage of the self-driving, cooperating cars and the number of the lanes

In Figure 4, we can have a look at the relation between the density and these two variables. Since the lower density shows better traffic condition, the self-driving, cooperating cars exactly do good to the traffic condition. We found that the effects of this new kind of cars increase almost linear from 10% to 100% for lanes 3 and 4. For 2 lanes, the difference between 90% and 100% makes tiny efforts to the density. For 5 lanes, there is a tipping point where performance changes markedly from 90% to 100%. This figure shows in 2-lane-road and 5-lane-road the density is lower while in 3-lane-road and 4-lane-road the density is higher.

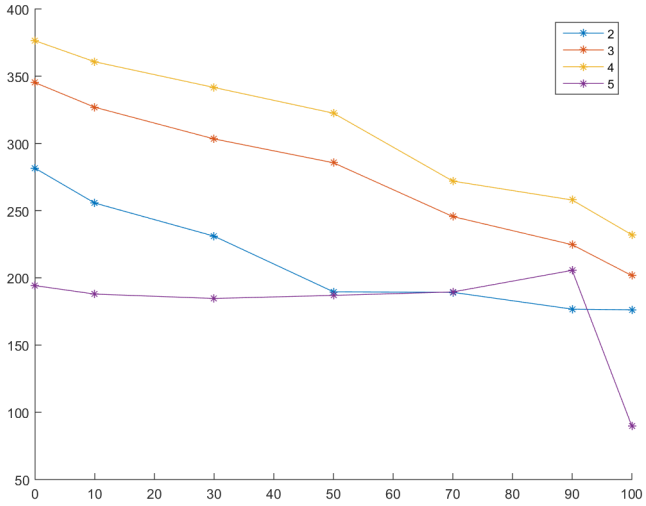


Figure 4: Relation between the density and the percentage of the self-driving, cooperating cars and the number of the lanes

#### 4.1.3 Peak and/or average traffic volume

### 4.2 Optimal situation investigation

#### 4.2.1 Comprehensive Evaluation Index

In order to examine under what conditions, a lane should be dedicated to self-driving cars, we first need a specific comprehensive evaluation index to quantify the traffic flow, and then we can compare the variations after we reserve one lane exclusively for self-driving cars and determine whether this action is beneficial. Since we have quantitative information about the traffic flow (i.e. the average speed, density and flow), we dont need to rely on Analytic Hierarchy Process (AHP), which is relatively subjective. We implement fuzzy synthetic evaluation (FSE)[**?**]. FSE is a multiple-criteria-decision-making method proposed by American mathematician and computer scientist Lotfi A. Zadeh. It can determine the weights of each variable and obtain one uniquely determined weighted score of each situation. We use this technique to analyze the performance under different scenarios.

Below we use the traffic flow at peak hours when there are 50% self-driving vehicles as an example. Other situations can be processed in a similar way.

**Collect attributes data for all variables**

Three variables are considered in our model: average speed, density and flow. The variables are the number of lanes: 2, 3, 4 and 5. The values of each attribute of each variable are listed in the table below.

All the data of B and H are listed in the table below.

| # of lanes | $v$ | $\rho$ | $q$ |
|---|---|---|---|
| 2 | 28.13 | 2670.12 | 189.61 |
| 3 | 40.76 | 4896.78 | 285.58 |
| 4 | 41.41 | 7201.30 | 322.48 |
| 5 | 51.50 | 9870.94 | 186.96 |

Table 3: Attributes data for all variables.

## 5 Sensitivity Analysis

### 5.1 Probability of Random Deceleration

In the simulation, we suppose the probability of random deceleration is 0.3. Indeed, we find little information about it. Since the random deceleration depends on human behavior which may be influenced by people's feeling, weather, surroundings and many other uncontrollable factors, it is hard to be

determined and adapt to all the circumstances. If the probability is raised, it adds to the traffic condition and vice versa.

### 5.2 Choice of Acceleration

There are two kinds of acceleration in this model. One is the maximum acceleration which is used for calculation of the stopping distance. This acceleration is achieved by information from website[1]. However, it is still an approximate value which affected by the performance of the vehicles, the road condition like down hill or snow and ice. Another acceleration is the usual acceleration which is randomly chosen from 5 to 7 $mile \cdot h \cdot s$. This is also determined from the website[4]. In this model, the choice of acceleration and deceleration is the same, but in fact, acceleration is related to the motor while deceleration is related to the brake. If the acceleration is raised, it helps to increase the traffic flow.

### 5.3 Reaction Time

The reaction time ranges from 0.5 to 1.25 second.3.2.1 To make an overall consideration, we choose 1 second as the reaction time. The longer reaction time leads to harder traffic condition.

## 6 Strengths and Weaknesses

### 6.1 Strengths

- **Considering various kinds of factors.** Our model takes the effects on traffic flow of the number of lanes, peak and average traffic volume and percentage of self-driving, cooperating vehicles into account.
- **Analyzing large amount of data.** Our model analyzes the traffic condition on Interstates 5, 90 and 405, as well as State Route 520. Hundreds of sections of highway that each section is road between two Milepost and tens of millions of vehicles are simulated.
- **Making detailed analysis.** Our model provides information about traffic flow, average velocity and traffic density of sections of the highway. Besides the spatiotemporal patterns histograms and line charts, we also use FSE to combine the three criteria and produce a new criterion Comprehensive Evaluation Index to show the comprehensive evaluation of the traffic condition under different circumstances. These analysis bring about an optimization method to enlarge traffic flow and improve traffic.

### 6.2 Weakness

- **No lane changing.** Due to time limit, we have not taken lane changing into account which means there is no overtaking in the whole process no matter for self-driving or human-driving vehicles. Overtaking may aggravate the traffic condition in the busy traffic. More seriously, the new-entering cars won't change their lanes which means the entering lane may be much more busy than other lanes if the traffic flow is huge.
- **Cancelling between the entering and leaving cars.** Since we do not have the number for the entering and leaving cars on each section of the highway, we simply consider that the traffic counts of one section is the sum of the previous section and the entering ones or the difference between the previous one and the leaving ones. This means some traffic volume is cancelled under this hypothesis. These traffic volume makes the traffic worse in the real world.

# References

[1] Neilsen, Joel. "Stopping Distance", Safe Drive Training, <http://sdt.com.au/safedrive-directory-STOPPINGDISTANCE.htm>. Accessed 21 January 2017.

[2] Shiffman, Daniel. "Cellular Automata", *The Nature of Code*, Lightning Source Inc, 2008. 323-354. Print.

[3] Stromberg, Joseph. "Why do traffic jams sometimes form for no reason?", Vox, <http://www.vox.com/2014/11/24/7276027/traffic-jam>. Accessed 22 January 2017.

[4] Consumer Reports, "Best & Worst Car Acceleration", Consumer Reports, <http://www.consumerreports.org/cro/news/2012/05/best-worst-acceleration/index.htm>. Accessed 23 January 2017.

# Appendix

## CMakeLists.txt

```
cmake_minimum_required(VERSION 3.5)
project(MCM2017)

set(CMAKE_CXX_STANDARD 14)
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}/
    bin)

set(SOURCE_FILES src/main.cpp src/Layout.cpp src/Car.cpp src/
    HumanCar.cpp src/AutoCar.cpp)
add_executable(MCM2017 ${SOURCE_FILES})
```

## C++ Code

```
/*************************************************
mcm.h
*************************************************/

#ifndef MCM2017_MCM_H
#define MCM2017_MCM_H

using namespace std;

// STD LIBS
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <time.h>

// IO Stream
#include <iostream>
#include <fstream>
#include <sstream>
#include <iomanip>

// STL
#include <string>
#include <vector>
#include <list>
#include <queue>
#include <unordered_map>
#include <algorithm>

#define DIR_ASC true
#define DIR_DESC false
#define TIME_NORMAL true
#define TIME_BUSY false

class Layout;

class Car;

class HumanCar;
```

```
40
41  class AutoCar;
42
43  const double BRAKE_ACCERLATION = 81576.5927;
44
45  template<typename T1, typename T2>
46  static ostream &operator<<(ostream &os, const std::pair<T1, T2> &
        T)
47  {
48      return os << "pair(" << T.first << "," << T.second << ")";
49  };
50
51  struct Data
52  {
53      int routeId;
54      double start, end;
55      int traffic;
56      string type;
57      int paneDESC, paneASC;
58  };
59
60  /**
61   *
62   * @param hourCount The number of cars going through per hour
63   * @param paneNum The number of the panes on the roud
64   * @return pair<double v, double s>
65   * v is the average speed in mile/h
66   * s is the distance between two heads of the cars
67   */
68  static pair<double, double> getIdealSpeedDistance(double
        hourCount, int paneNum)
69  {
70      // const double a = 97104;
71      const double carLength = 16. / 5280.;
72      const double maxV = 60.;
73      double A = hourCount / (2 * BRAKE_ACCERLATION);
74      double B = hourCount / 3600. - paneNum;
75      double C = carLength * hourCount;
76      double Delta = B * B - 2 * A * C;
77      if (Delta <= 0) return pair<double, double>(hourCount /
            paneNum * 20. / 5280., 20. / 5280.);
78      double v = (-B + sqrt(Delta)) / (2 * A);
79      v = min(v, maxV);
80      double s = v * v / (2 * BRAKE_ACCERLATION) + v / 3600. +
            carLength;
81      return pair<double, double>(v, s);
82  }
83
84  #endif //MCM2017_MCM_H
```

```
1  /*************************************************
2  Car.h
3  *************************************************/
4
5  #ifndef MCM2017_CAR_H
6  #define MCM2017_CAR_H
7
8  #include "mcm.h"
```

```

class Car
{
public:
    enum STATE
    {
        UNIFORM = 0, ACCELERATE = 1, DECELERATE = -1, BRAKE = -2
    };

    double m_respondTime, m_speed, m_pos, m_acceleration;

    int m_wait;

    double m_distance;

    bool m_deleteFlag;

    STATE m_state, m_stateNext;

    Car *m_frontCar, *m_backCar;

    int m_milepostNo;

    Layout *m_layout;

    Car(Layout *layout, int milepostNo);

    virtual ~Car() = 0;

    virtual void move(double period);

    virtual double getAcceleration() = 0;

    inline double idealDistance(double period)
    {
        if (m_speed < 20.)return 20. / 5280.;
        return m_speed * m_speed / (2 * BRAKE_ACCERLATION)
               + m_speed * m_respondTime / 3600. + 16. / 5280.;
    }
};


#endif //MCM2017_CAR_H
```

```
/*************************************************
Car.cpp
*************************************************/

#include "Car.h"
#include "Layout.h"

Car::Car(Layout *layout, int milepostNo)
{
    m_layout = layout;
    m_milepostNo = milepostNo;
    m_speed = 0;
    //m_acceleration = 6.;
    m_acceleration = 2.5 + 0.5 * ((double) rand()) / RAND_MAX;
```

```
15      m_wait = 0;
16      m_state = UNIFORM;
17      m_stateNext = UNIFORM;
18      m_distance = 0;
19      m_deleteFlag = false;
20      m_frontCar = m_backCar = NULL;
21  }
22
23  Car::~Car()
24  {
25
26  }
27
28  void Car::move(double period)
29  {
30      if (m_wait >= 0)
31      {
32          m_wait--;
33      }
34      else
35      {
36          auto d = idealDistance(period);
37          if (d * 1.1 < m_distance)
38          {
39              m_wait = int(m_respondTime / period);
40              m_stateNext = ACCELERATE;
41          }
42          else if (d * 0.9 > m_distance)
43          {
44              m_wait = int(m_respondTime / period);
45              m_stateNext = DECELERATE;
46          }
47      }
48      if (m_wait < 0 && m_state != m_stateNext)
49      {
50          m_state = m_stateNext;
51      }
52      auto newSpeed = max(0., min(60., m_speed + getAcceleration()
            * period));
53      auto newPos = m_pos - 0.5 * (m_speed + newSpeed) * period /
            3600.;
54      if (m_frontCar)
55      {
56          if (m_milepostNo != m_frontCar->m_milepostNo)
57          {
58              double mile = 0;
59              for (int i = m_frontCar->m_milepostNo; i <
                    m_milepostNo; i++)
60              {
61                  mile += m_layout->m_milepost[i].mile;
62              }
63              if (newPos + (mile - m_frontCar->m_pos) <= 20. /
                    5280.)
64              {
65                  newPos = -(mile - m_frontCar->m_pos) + 20. /
                        5280.;
66                  newSpeed = (m_pos - newPos) / (period / 3600.);
67                  //cout << "test1\t" << newPos << "\t" << newSpeed
```

```
                    << endl;
68              }
69          }
70          else if (newPos - m_frontCar->m_pos <= 20. / 5280.)
71          {
72              newPos = m_frontCar->m_pos + 20. / 5280.;
73              newSpeed = (m_pos - newPos) / (period / 3600.);
74
75          }
76      }
77      if (newPos >= m_layout->m_milepost[m_milepostNo].mile)
78      {
79          //cout << m_milepostNo << "\t" << m_pos << "\t" <<
                m_frontCar->m_pos << "\t" << newPos << "\t" <<
                newSpeed << endl;
80      }
81      m_pos = newPos;
82      m_speed = newSpeed;
83      /*if (newSpeed < 1e-5)
84      {
85          cout << m_milepostNo << "\t" << m_pos << "\t" <<
                m_frontCar->m_pos << "\t" << newPos << "\t" <<
                newSpeed << endl;
86      }*/
87  }
```

```
1  /************************************************
2  HumanCar.h
3  ************************************************/
4
5  #ifndef MCM2017_HUMANCAR_H
6  #define MCM2017_HUMANCAR_H
7
8  #include "Car.h"
9
10 class HumanCar : public Car
11 {
12 public:
13     HumanCar(Layout *layout, int milepostNo);
14
15     ~HumanCar();
16
17     double getAcceleration();
18
19 };
20
21
22 #endif //MCM2017_HUMANCAR_H
```

```
1  /************************************************
2  HumanCar.cpp
3  ************************************************/
4
5  #include "HumanCar.h"
6
7  HumanCar::HumanCar(Layout *layout, int milepostNo) : Car(layout,
       milepostNo)
```

```
8  {
9      m_respondTime = 1.;
10 }
11
12 HumanCar::~HumanCar()
13 {
14
15 }
16
17 double HumanCar::getAcceleration()
18 {
19     return m_acceleration * m_state;
20 }
```

```
1  /*************************************************
2  AutoCar.h
3  *************************************************/
4
5  #ifndef MCM2017_AUTOCAR_H
6  #define MCM2017_AUTOCAR_H
7
8  #include "Car.h"
9
10 class AutoCar : public Car
11 {
12 public:
13     AutoCar(Layout *layout, int milepostNo);
14
15     ~AutoCar();
16
17     double getAcceleration();
18
19     void move(double period);
20
21     bool m_chain;
22 };
23
24
25 #endif //MCM2017_AUTOCAR_H
```

```
1  /*************************************************
2  AutoCar.cpp
3  *************************************************/
4
5  #include "AutoCar.h"
6  #include "Layout.h"
7
8  AutoCar::AutoCar(Layout *layout, int milepostNo) : Car(layout,
       milepostNo)
9  {
10     m_respondTime = 0;
11     m_chain = false;
12 }
13
14 AutoCar::~AutoCar()
15 {
16
```

```
17  }
18
19  double AutoCar::getAcceleration()
20  {
21      return m_acceleration * m_state;
22  }
23
24  void AutoCar::move(double period)
25  {
26      if (m_frontCar)
27      {
28          double deltaPos = 0;
29          if (m_distance > idealDistance(period) * 1.1)
30          {
31              m_chain = false;
32              Car::move(period);
33              return;
34          }
35
36          if (m_speed > 20 && m_speed <= 40 && m_distance >= 20. /
                5280. && m_distance <= 70. / 5280)
37          {
38              m_chain = true;
39              deltaPos = 45. / 5280.;
40          }
41          else if (m_speed > 40 && m_distance >= 65. / 5280)
42          {
43              m_chain = true;
44              deltaPos = 100. / 5280.;
45          }
46
47          if (m_chain)
48          {
49              double newPos = 0;
50              if (m_milepostNo != m_frontCar->m_milepostNo)
51              {
52                  double mile = 0;
53                  for (int i = m_frontCar->m_milepostNo; i <
                        m_milepostNo; i++)
54                  {
55                      mile += m_layout->m_milepost[i].mile;
56                  }
57                  newPos = -(mile - m_frontCar->m_pos) + deltaPos;
58              }
59              else
60              {
61                  newPos = m_frontCar->m_pos + deltaPos;
62              }
63              m_speed = m_frontCar->m_speed;
64              m_pos = newPos;
65              return;
66          }
67
68      }
69      else
70      {
71          m_chain = false;
72      }
```

```
    Car::move(period);
}
```

```
/*************************************************
Layout.h
*************************************************/

#ifndef MCM2017_LAYOUT_H
#define MCM2017_LAYOUT_H

#include "mcm.h"
#include "Car.h"
#include "HumanCar.h"
#include "AutoCar.h"

class Layout
{
public:
    struct stat
    {
        double speedPeriod, carTotal;
        int carPeriod, carFlow;
    };

    struct Milepost
    {
        double idealSpeed, idealDistance, idealPeriod, idealCount
            ;
        double mile, lastEnterTime;
        std::vector<stat> statData;
        std::vector<std::list<Car *> > cars;
    };

    std::vector<Milepost> m_milepost;
    double m_period, m_autoPercentage;

    std::string m_outputPath;

    Layout(vector <Data> m_data, bool isNormal, bool isASC,
        double autoPercentage);

    Car *addCar(std::list<Car *> &carQueue, double speed, double
        pos, int milepostNo);

    void simulate(double time = 3600.);

    ofstream m_speed_file;

    bool openFile();

    void closeFile();

    void printSpeed(double interval);

};


#endif //MCM2017_LAYOUT_H
```

```
/*************************************************
Layout.cpp
*************************************************/

#include "Layout.h"

Layout::Layout(vector <Data> m_data, bool isNormal, bool isASC,
    double autoPercentage)
{
    m_autoPercentage = autoPercentage;

    double offsetDistance = 0.;
    if (isASC)
    {
        std::reverse(m_data.begin(), m_data.end());
    }

    for (int i = 0; i < m_data.size(); i++)
    {
        int paneNum = isASC ? m_data[i].paneASC : m_data[i].
            paneDESC;
        double traffic = m_data[i].traffic * (isNormal ? 0.02 :
            0.04);
        auto speedDistance = getIdealSpeedDistance(traffic,
            paneNum);
        //cout << i << '\t' << speedDistance << '\t';

        m_milepost.push_back(Milepost());
        Milepost &data = m_milepost.back();

        data.idealSpeed = speedDistance.first;
        data.idealDistance = speedDistance.second;
        data.mile = m_data[i].end - m_data[i].start;

        //


        if (offsetDistance > data.idealDistance)
        {
            offsetDistance = 0;
        }

        data.idealCount = (data.mile + offsetDistance) / data.
            idealDistance;
        data.idealPeriod = data.idealDistance / data.idealSpeed *
            3600; // in seconds
        data.lastEnterTime = 0;

        for (int j = 0; j < paneNum; j++)
        {
            std::list<Car *> carList;
            Car *frontCar = NULL;
            for (int k = 1; k <= data.idealCount; k++)
            {
                auto newCar = addCar(carList, data.idealSpeed, k
                    * data.idealDistance - offsetDistance, i);
```

```
48                  if (k == 1)
49                  {
50                      if (i == 0) frontCar = NULL;
51                      else if (m_milepost[i - 1].cars.size() <= j)
                            frontCar = NULL;
52                      else frontCar = m_milepost[i - 1].cars[j].
                            back();
53                  }
54                  if (frontCar)frontCar->m_backCar = newCar;
55                  newCar->m_frontCar = frontCar;
56                  frontCar = newCar;
57
58                  //cout << carList.back()->m_pos << '\t';
59              }
60              //cout << j << "\t" << carList.size() << endl;
61              data.cars.push_back(carList);
62              data.statData.push_back(stat());
63          }
64
65
66          //cout << data.cars[0].size() << '\t';
67
68
69          //
70          offsetDistance += data.mile - ((int) data.idealCount) *
               data.idealDistance;
71
72          //cout << endl;
73
74      }
75
76  }
77
78  Car *Layout::addCar(std::list<Car *> &carQueue, double speed,
        double pos, int milepostNo)
79  {
80      Car *car;
81      if (((double) rand()) / RAND_MAX < m_autoPercentage)
82      {
83          car = new AutoCar(this, milepostNo);
84      }
85      else
86      {
87          car = new HumanCar(this, milepostNo);
88      }
89      car->m_speed = speed;
90      car->m_pos = pos;
91      carQueue.push_back(car);
92      return car;
93  }
94
95  void Layout::simulate(double time)
96  {
97      int maxI = int(time / m_period + 1);
98      for (int i = 0; i < 51; i++)
99      {
100         cout << '*';
101     }
```

```
102     cout << endl << '>';
103
104     for (int i = 0; i <= maxI; i++)
105     {
106         double now = i * m_period;
107         for (int j = 0; j < m_milepost.size(); j++)
108         {
109             for (int k = 0; k < m_milepost[j].cars.size(); k++)
110             {
111                 //
112                 auto it = m_milepost[j].cars[k].begin();
113                 if (it == m_milepost[j].cars[k].end())continue;
114                 (*it)->m_deleteFlag = false;
115                 double distance;
116                 if (j == 0)
117                 {
118                     (*it)->m_deleteFlag = true;
119                     distance = (*it)->idealDistance(m_period);
120                 }
121                 else if (m_milepost[j - 1].cars.size() <= k)
122                 {
123                     (*it)->m_deleteFlag = true;
124                     distance = (*it)->idealDistance(m_period);
125                 }
126                 else if (m_milepost[j - 1].cars[k].size() == 0)
127                 {
128                     distance = (*it)->idealDistance(m_period);
129                 }
130                 else
131                 {
132                     auto front = m_milepost[j - 1].cars[k].back()
                            ;
133                     distance = m_milepost[j - 1].mile
134                                 - front->m_pos + (*it)->m_pos;
135                     //cout << j << '\t' << distance << '\t' <<
                        m_milepost[j - 1].mile << '\t' << front->
                        m_pos << '\t' << (*it)->m_pos << '\t' <<
                        endl;
136                 }
137                 //(*it)->move(distance, m_period);
138                 (*it)->m_distance = distance;
139                 /*if ((*it)->m_distance <= 0)
140                 {
141                     cout << i << "\t" << j << "\t" << k << "\t"
142                          << 1 << "/" << m_milepost[j].cars[k].
                                size()
143                          << "\t" << (*it)->m_distance << endl;
144                     auto car = (*it);
145                     cout << m_milepost[j].mile << endl;
146                     for (int i = 0; i < 10; i++)
147                     {
148                         if (!car->m_frontCar)break;
149                         cout << car->m_milepostNo << "\t" << car
                                ->m_pos << endl;
150                         car = car->m_frontCar;
151                     }
152                 }*/
153
```

```
154
155                 //
156                 if (m_milepost[j].cars[k].size() > 1)
157                 {
158                     int index = 1;
159                     for (double lastPos = (*it++)->m_pos; it !=
                            m_milepost[j].cars[k].end(); ++it)
160                     {
161                         //(*it)->move((*it)->m_pos - lastPos,
                                m_period);
162                         (*it)->m_distance = (*it)->m_pos -
                                lastPos;
163                         /*if ((*it)->m_distance <= 0)
164                         {
165                             cout << i << "\t" << j << "\t" << k
                                    << "\t"
166                                 << index + 1 << "/" <<
                                        m_milepost[j].cars[k].size()
167                                 << "\t" << (*it)->m_distance <<
                                        endl;
168                             auto car = (*it);
169                             cout << m_milepost[j].mile << endl;
170                             for (int i = 0; i < 10; i++)
171                             {
172                                 if (!car->m_frontCar)break;
173                                 cout << car->m_milepostNo << "\t"
                                        << car->m_pos << endl;
174                                 car = car->m_frontCar;
175                             }
176                         }*/
177                         /*if ((*it)->m_pos > m_milepost[j].mile)
178                         {
179                             cout << i << "\t" << j << "\t" << k
                                    << "\t"
180                                 << index + 1 << "/" <<
                                        m_milepost[j].cars[k].size()
181                                 << "\t" << (*it)->m_pos << "/"
                                    << m_milepost[j].mile << endl
                                    ;
182                         }*/
183                         lastPos = (*it)->m_pos;
184                         index++;
185                     }
186                 }
187             }
188         }
189         for (int j = 0; j < m_milepost.size(); j++)
190         {
191             bool addCarFlag = false;
192             if (now - m_milepost[j].lastEnterTime >= m_milepost[j
                    ].idealPeriod)
193             {
194                 m_milepost[j].lastEnterTime += m_milepost[j].
                        idealPeriod;
195                 addCarFlag = true;
196             }
197             for (int k = 0; k < m_milepost[j].cars.size(); k++)
198             {
```

```
199                     //
200                     for (auto it:m_milepost[j].cars[k])
201                     {
202                         //cout  << i << setw(4) << j << endl;
203                         //auto speed = it->m_speed;
204                         it->move(m_period);
205                         m_milepost[j].statData[k].speedPeriod += it->
                                m_speed;
206                         m_milepost[j].statData[k].carPeriod++;
207                         /*if (j == 66 && abs(it->m_speed) <= 1e-5)
208                         {
209                             cout << it->m_distance << "\t" << it->
                                    m_speed << "\t" << speed << "\t" << it
                                    ->m_pos << "\t"
210                                  << it->m_state
211                                  << endl;
212                         }*/
213                     }
214
215                     //
216                     auto front = m_milepost[j].cars[k].front();
217                     if (!front)
218                     {
219                         //cout << i << "\t" << j << "\t" << k << "\t"
                                << m_milepost[j].cars[k].size() << endl;
220                     }
221                     else if (front->m_pos < 0)
222                     {
223                         m_milepost[j].cars[k].pop_front();
224                         if (!front->m_frontCar)
225                         {
226                             if (front->m_backCar)front->m_backCar->
                                    m_frontCar = NULL;
227                             delete front;
228                         }
229                         else
230                         {
231                             m_milepost[j - 1].cars[k].push_back(front
                                    );
232                             m_milepost[j - 1].statData[k].carFlow++;
233                             front->m_pos += m_milepost[j - 1].mile;
234                             if (front->m_milepostNo != j)
235                             {
236                                 cout << j << endl;
237                             }
238                             front->m_milepostNo = j - 1;
239                         }
240                     }
241
242                     //
243                     if (addCarFlag)
244                     {
245                         bool newCarFlag = false;
246                         if (j == m_milepost.size() - 1)newCarFlag =
                                true;
247                         if (m_milepost[j + 1].cars.size() <= k)
                                newCarFlag = true;
248                         if (newCarFlag)
```

```
249                         {
250                             auto frontCar = m_milepost[j].cars[k].
                                    back();
251                             auto pos = m_milepost[j].mile;
252                             if (frontCar)
253                             {
254                                 if (frontCar->m_pos + 20. / 5280. >
                                        m_milepost[j].mile)
255                                 {
256                                     pos = frontCar->m_pos + 21. /
                                            5280.;
257                                 }
258                             }
259                             auto newCar = addCar(m_milepost[j].cars[k
                                    ],
260                                     m_milepost[j].idealSpeed, pos, j)
                                            ;
261                             m_milepost[j].statData[k].carFlow++;
262                             if (frontCar) newCar->m_backCar =
                                    frontCar->m_backCar;
263                             if (newCar->m_backCar)newCar->m_backCar->
                                    m_frontCar = newCar;
264                             newCar->m_frontCar = frontCar;
265                             if (frontCar) frontCar->m_backCar =
                                    newCar;
266                         }
267                     }
268
269                     m_milepost[j].statData[k].carTotal += m_milepost[
                            j].cars[k].size();
270                 }
271             }
272
273
274             if (i % (maxI / 50) == 0)
275             {
276                 //cout << now << "/" << time << endl;
277                 cout << "\b->" << flush;
278                 //printSpeed(600);
279             }
280
281         }
282     }
283
284     bool Layout::openFile()
285     {
286         m_speed_file.open(m_outputPath);
287         if (!m_speed_file.is_open())
288         {
289             cerr << "Outpur File Failed to Open!" << endl <<
                        m_outputPath;
290             return false;
291         }
292         return true;
293     }
294
295     void Layout::closeFile()
296     {
```

```
297     m_speed_file.close();
298 }
299
300 void Layout::printSpeed(double interval)
301 {
302     for (int i = 0; i < m_milepost.size(); i++)
303     {
304         for (int j = 0; j < m_milepost[i].cars.size(); j++)
305         {
306             double speedAvg = m_milepost[i].statData[j].carPeriod
                    > 0 ?
307                                 m_milepost[i].statData[j].
                                       speedPeriod / m_milepost[i].
                                       statData[j].carPeriod : 0;
308             double carFlow = m_milepost[i].statData[j].carFlow;
309             double carDensity = m_milepost[i].statData[j].
                   carTotal / interval / m_milepost[i].mile;
310             m_speed_file << i << "\t" << j << "\t"
311                          << speedAvg << "\t"
312                          << carFlow << "\t"
313                          << carDensity << "\t"
314                          << endl;
315         }
316     }
317     m_speed_file << endl;
318     cout << endl;
319 }
```

```
1  /*************************************************
2  main.cpp
3  *************************************************/
4
5  #include "mcm.h"
6  #include "Layout.h"
7
8  const double period = 0.1;
9  double totalTime = 0;
10 string name = "";
11
12 int simulate(const std::vector<Data> &m_data, bool isNormal, bool
       isASC, string autoPercentage)
13 {
14     Layout lay(m_data, isNormal, isASC, atof(autoPercentage.c_str
           ()));
15     lay.m_period = period;
16     lay.m_outputPath = "output/";
17     lay.m_outputPath += name + "_";
18     lay.m_outputPath += (isNormal ? "norm_" : "busy_");
19     lay.m_outputPath += (isASC ? "asc_" : "desc_");
20     lay.m_outputPath += autoPercentage;
21     lay.m_outputPath += ".txt";
22     cout << lay.m_outputPath << endl;
23     if (!lay.openFile())return -1;
24     lay.simulate(totalTime);
25     lay.printSpeed(totalTime / period);
26     lay.closeFile();
27     return 0;
28 }
```

```
29
30 int main(int argc, char *argv[])
31 {
32     const int MAX_ARG = 3;
33     string args[MAX_ARG] = {"5", "3600", ""};
34     for (int i = 1; i <= min(argc - 1, MAX_ARG); i++)
35     {
36         args[i - 1] = argv[i];
37     }
38
39     for (int i = 0; i < MAX_ARG; i++)
40     {
41         cout << args[i] << "\t";
42     }
43     cout << (args[2] == "");
44     cout << endl;
45
46     name = args[0];
47     string filename = "data/" + name + ".txt";
48     totalTime = atof(args[1].c_str());
49
50     srand((unsigned) time(NULL));
51
52     ifstream in(filename);
53
54     if (!in.is_open())
55     {
56         cerr << "File not found!" << endl;
57         return -1;
58     }
59
60     std::vector<Data> m_data;
61     while (!in.eof())
62     {
63         string str;
64         getline(in, str);
65         istringstream ss(str);
66         if (ss.eof())continue;
67         Data data;
68         data.routeId = atoi(name.c_str());
69         ss >> data.start >> data.end >> data.traffic
70            >> data.type >> data.paneDESC >> data.paneASC;
71         m_data.push_back(data);
72     }
73
74     cout << m_data.size() << endl;
75
76     std::vector<string> percentVec = {"0.0", "0.1", "0.3", "0.5",
         "0.7", "0.9", "1.0"};
77
78     for (auto autoPercentage : percentVec)
79     {
80         if (args[2] == "" || args[2] == "0")
81         {
82             simulate(m_data, false, false, autoPercentage);
83         }
84         if (args[2] == "" || args[2] == "1")
85         {
```

```
86              simulate(m_data, false, true, autoPercentage);
87          }
88          if (args[2] == "" || args[2] == "2")
89          {
90              simulate(m_data, true, false, autoPercentage);
91              simulate(m_data, true, true, autoPercentage);
92          }
93      }
94
95
96      return 0;
97  }
```

## MATLAB Code

```
1  %************************************************
2  % readData.m
3  %************************************************
4
5  function A=readData(route)
6      order_list=['asc ';'desc'];
7      type_list=['norm';'busy'];
8      auto_list=['0.0';'0.1';'0.3';'0.5';'0.7';'0.9';'1.0'];
9
10     for i=1:2
11         for j=1:2
12             for k=1:7
13                 if(j==1)
14                     order='asc';
15                 else
16                     order='desc';
17                 end
18                 str=['../output/',route,'_',type_list(i,:),'_',
                       order,'_',auto_list(k,:),'.txt'];
19                 temp=textread(str);
20                 s=size(temp);
21                 A(i,j,k,1:s(1),1:5)=temp(1:s(1),1:5);
22             end
23         end
24     end
25 end
```

```
1  %************************************************
2  % parseData.m
3  %************************************************
4
5  function [result,mileSumArr,mileNumArr] = parseData( route,n )
6      D=readData(route);
7      result=[];
8
9      for percentage=1:7
10
11         mileSumArr=zeros(2,5);
12         mileNumArr=zeros(2,5);
13
14         for aord=1:2
```

```
15
16              averageData=[];
17
18              for i=1:5
19                  averageData(i).speed=0;
20                  averageData(i).flow=0;
21                  averageData(i).density=0;
22              end
23
24
25              DATA(:,:)=D(n,aord,percentage,:,:);
26
27              sizeDATA=size(DATA);
28
29              [milepostStart,milepostEnd]=textread(['../data/',
                    route,'.txt'],'%f%f%*[^\n]');
30              milepost=zeros(1,length(milepostStart));
31              milepost(:)=milepostEnd(:)-milepostStart(:);
32
33
34              mileSum=sum(milepost);
35              mileSumPath=zeros(1,5);
36              mileNumPath=zeros(1,5);
37
38              milepostNo=1;
39              milepostTotal=sizeDATA(1);
40
41              for i=1:length(milepost)
42                  milepostData(i).speed=0;
43                  milepostData(i).flow=0;
44                  milepostData(i).density=0;
45                  milepostData(i).pathNum=0;
46              end
47
48              for i=1:milepostTotal
49
50                  if(DATA(i,1)+1~=milepostNo)
51                      break;
52                  end
53
54                  milepostData(milepostNo).pathNum=milepostData(
                        milepostNo).pathNum+1;
55
56                  endFlag=0;
57                  if(i==milepostTotal)
58                      endFlag=1;
59                  elseif(DATA(i+1,2)==0)
60                      endFlag=1;
61                  end
62
63                  milepostData(milepostNo).speed=milepostData(
                        milepostNo).speed+DATA(i,3);
64                  milepostData(milepostNo).flow=milepostData(
                        milepostNo).flow+DATA(i,4);
65                  milepostData(milepostNo).density=milepostData(
                        milepostNo).density+DATA(i,5);
66
67                  if(endFlag)
```

```
                    pathNum=milepostData(milepostNo).pathNum;
                    mileSumPath(pathNum)=mileSumPath(pathNum)+
                        milepost(milepostNo);
                    mileNumPath(pathNum)=mileNumPath(pathNum)+1;
                    milepostNo=milepostNo+1;
                end
            end

            for i=1:length(milepost)
                pathNum=milepostData(i).pathNum;
                averageData(pathNum).speed = averageData(pathNum)
                    .speed + ...
                    milepostData(i).speed / pathNum * (milepost(i
                        ) / mileSumPath(pathNum));
                averageData(pathNum).flow = averageData(pathNum).
                    flow + ...
                    milepostData(i).flow * (milepost(i) /
                        mileSumPath(pathNum));
                averageData(pathNum).density = averageData(
                    pathNum).density + ...
                    milepostData(i).density / mileNumPath(pathNum
                        );

                averageData(1).speed = averageData(1).speed + ...
                    milepostData(i).speed / pathNum * (milepost(i
                        ) / mileSum);
                averageData(1).flow = averageData(1).flow + ...
                    milepostData(i).flow * (milepost(i) / mileSum
                        );
                averageData(1).density = averageData(1).density +
                     ...
                    milepostData(i).density / length(milepost);
            end
            for i=1:5
                mileSumArr(aord,i)=mileSumPath(i);
                mileNumArr(aord,i)=mileNumPath(i);
                if(aord==1)
                    result(percentage,i).speed=averageData(i).
                        speed;
                    result(percentage,i).flow=averageData(i).flow
                        ;
                    result(percentage,i).density=averageData(i).
                        density;
                else
                    if(mileSumArr(1,i) + mileSumArr(2,i)>0)
                        result(percentage,i).speed = ...
                            result(percentage,i).speed * ...
                            (mileSumArr(1,i) / (mileSumArr(1,i) +
                                mileSumArr(2,i))) + ...
                            averageData(i).speed * ...
                            (mileSumArr(2,i) / (mileSumArr(1,i) +
                                mileSumArr(2,i)));

                        result(percentage,i).flow = ...
                            result(percentage,i).flow * ...
                            (mileSumArr(1,i) / (mileSumArr(1,i) +
                                mileSumArr(2,i))) + ...
                            averageData(i).flow * ...
```

```
                                (mileSumArr(2,i) / (mileSumArr(1,i) +
                                    mileSumArr(2,i)));
                        end
                        if(mileNumArr(1,i) + mileNumArr(2,i)>0)
                            result(percentage,i).density = ...
                                result(percentage,i).density * ...
                                (mileNumArr(1,i) / (mileNumArr(1,i) +
                                    mileNumArr(2,i))) + ...
                                averageData(i).density * ...
                                (mileNumArr(2,i) / (mileNumArr(1,i) +
                                    mileNumArr(2,i)));
                        end
                    end
                end
            end

        end

    end
```

```
%************************************************
% drawSingle.m
%************************************************

function drawSingle(result,filepath)
    for i=1:3
        figure(i);
        clf;
    end
    xValue=[0,10,30,50,70,90,100];
    yValue=zeros(3,7);
    ylegend=[];
    for pathNum=2:5
        for percentage=1:7
            yValue(1,percentage)=result(percentage,pathNum).speed
                ;
            yValue(2,percentage)=result(percentage,pathNum).flow;
            yValue(3,percentage)=result(percentage,pathNum).
                density;
        end
        if(sum(yValue(1,:))>0)
            ylegend=[ylegend;num2str(pathNum)];
        end
        for i=1:3
            figure(i);
            if(sum(yValue(i,:))>0)
                hold on;
                plot(xValue,yValue(i,:),'-*');
                hold off;
            end
        end
    end
    for i=1:3
        figure(i);
        legend(ylegend);
        % @TODO Add more information


```

```
        saveas(gcf,[filepath,'_',num2str(i),'.png']);
    end

end
```

```
%************************************************
% drawBar.m
%************************************************

function drawBar(result,result2,filepath)
    xValue=[0,10,30,50,70,90,100];
    yValue=zeros(6,7);
    for i=1:3
        figure(i);
        clf;
        xlim([-10 110]);
        set(gca,'XTick',0:10:100);
    end
    for pathNum=2:5
        for percentage=1:7
            yValue(1,percentage)=result(percentage,pathNum).speed
                ;
            yValue(2,percentage)=result2(percentage,pathNum).
                speed;
            yValue(3,percentage)=result(percentage,pathNum).flow;
            yValue(4,percentage)=result2(percentage,pathNum).flow
                ;
            yValue(5,percentage)=result(percentage,pathNum).
                density;
            yValue(6,percentage)=result2(percentage,pathNum).
                density;
        end
        for i=1:3
            figure(i);
            if(sum(yValue(i,:))>0)
                hold on;
                yBar=yValue(2*i-1:2*i,:);
                bar(xValue,yBar');
                hold off;
            end
        end
    end
    for i=1:3
        figure(i);
        legend('busy','normal');
        % @TODO Add more information


        saveas(gcf,[filepath,'_',num2str(i),'.png']);
    end

end
```

```
%************************************************
% main.m
%************************************************

```

```
clear all;
routes=[5,90,405,520];
for i=1:4
    [result,mileSumArr,mileNumArr]=parseData(num2str(routes(i)),
        2);
    drawSingle(result,['../graphs/',num2str(routes(i))]);

    results(i,:,:)=result(:,:);
    mileSumArrs(i,:)=mileSumArr(1,:)+mileSumArr(2,:);
    mileNumArrs(i,:)=mileNumArr(1,:)+mileNumArr(2,:);

end

for i=1:4
    result=parseData(num2str(routes(i)), 1);
    results(i+4,:,:)=result(:,:);
end

for percentage=1:7
    for path=1:5
        result(percentage,path).speed=0;
        result(percentage,path).flow=0;
        result(percentage,path).density=0;
        result2(percentage,path).speed=0;
        result2(percentage,path).flow=0;
        result2(percentage,path).density=0;
    end
    for path=2:5
        for i=1:4
            result(percentage,path).speed = result(percentage,
                path).speed + ...
                results(i,percentage,path).speed * (mileSumArrs(i
                    ,path) / sum(mileSumArrs(:,path)));
            result(percentage,path).flow = result(percentage,path
                ).flow + ...
                results(i,percentage,path).flow * (mileSumArrs(i,
                    path) / sum(mileSumArrs(:,path)));
            result(percentage,path).density = result(percentage,
                path).density + ...
                results(i,percentage,path).density * (mileNumArrs
                    (i,path) / sum(mileNumArrs(:,path)));
            result2(percentage,path).speed = result2(percentage,
                path).speed + ...
                results(i+4,percentage,path).speed * (mileSumArrs
                    (i,path) / sum(mileSumArrs(:,path)));
            result2(percentage,path).flow = result2(percentage,
                path).flow + ...
                results(i+4,percentage,path).flow * (mileSumArrs(
                    i,path) / sum(mileSumArrs(:,path)));
            result2(percentage,path).density = result2(percentage
                ,path).density + ...
                results(i+4,percentage,path).density * (
                    mileNumArrs(i,path) / sum(mileNumArrs(:,path))
                    );
        end
    end
end
drawSingle(result,'../graphs/all');
```

```
49
50 drawBar(result,result2,'../graphs/all_bar');
```