Georgia Tech School of Computer Science
College of Computing

Tutorial: Impact of incident to highway traffic flow analysis based on Cellular Automaton Approach

Ruxu Zhang, Ciyuan Yu, Han Gyol Kim

GitHub repository: https://github.gatech.edu/rzhang418/Project1

03/04/2019

1. Introduction

When it comes to traffic congestion, a car accident is a factor that cannot be ignored. In highways, due to high driving speed and relative high density of vehicles, car accidents always cause more serious consequences than accidents on normal roads do. This would inevitably lead to large impact to the traffic flow on the highways.

Our simulation model will focus on this topic, exploring the influence of car accidents to traffic flow on highway. We will choose the cellular automaton model and build a five-lane traffic system with lane changing and accidents. Then by establishing functions to generate "accidents" on the road, speed and density of traffic flow affected by accidents will be captured. Finally, the data to be collected from simulation will be visualized and plotted into diagram showing the relation between the traffic flow and passing time.

2. Methodology of Cellular Automaton Model

Traffic stream or phenomena is complex and nonlinear and defined as multi-dimensional traffic lanes with flow of vehicles over time. Vehicles follow each other on each lane, and they can choose lane changing when the former position on different lane is empty. CA model is one of the microscopic traffic models. In this model, a segment of roadway is made up of cells like the checkerboard and time is also discredited. Vehicles move from one cell to another. The first research using CA model for traffics simulation was conducted by *Nagel and Schreckenberg (1992)*, who simulated the single-lane highway traffic flow by a stochastic CA model. [2] By using CA model, we will develop a traffic flow simulator to evaluate dynamic traffic flow. We extend the existing CA models to describe the influence of a car accident in highway of five travel lanes of traffic flow model. We also add the lane changing rules to simulate the reality traffic condition. By simulation, we analyze all possible situations. The simulation will be implemented in C programming language.

Microscopic traffic flow models simulate single vehicle-driver units, based on driver's behavior. The dynamic variables of the models represent microscopic properties like the position and velocity of the vehicles. There are two modeling approach are known as Car-following model and Cellular automaton model. *Newell (2002)* establish the Car-following models which was used to determine how vehicles follow one another on a roadway. [3] *Kanai (2006)* propose a stochastic CA model for traffic flow and show the availability of CA modelling for the complex phenomena that occur in real traffic flow. [4] CA models describe the dynamical properties of the system in a discrete setting.

In details, the CA model dynamically includes three parts in our project, which are moving forward, switching lane and generating accidents. For the first part, there are four steps

movement in the simplest rule set, which leads to a realistic behavior, has been introduced in 1992 by *Nagel and Schreckenberg.*

Step 1. All the vehicles whose velocity has not reached the maximum Vmax will accelerate by one unit.
Step 2. Assume a car has m empty cells in front of it. If the velocity of the car (v) is biggerthan m, then the velocity becomes tom. If the velocity of the car (v) is smaller than m, then the velocity changes to v. ($v \rightarrow min[v, m]$)
Step 3. The velocity of the car may reduce by one unit with the probability p.
Step 4. After 3 steps, the new position of the vehicle can be determined by the current velocity and current position. ($xn' \rightarrow xn + vn$)

The following figure shows the four steps movements:

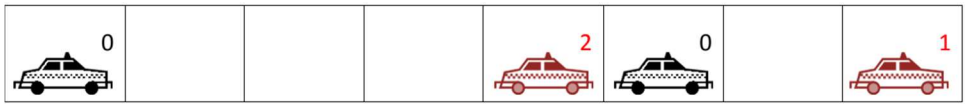
Previous Status:

Step1:

Step2:

Step3:

Step4:



After moving forward, the cars need to find if it is possible to run faster by switching the lane. This process can also be:

1. The distance ahead in current lane is smaller than the car speed.
2. The distance ahead in another lane is larger than in the current lane.
3. There exists an empty cell right in another lane.
4. The distance ahead of the following vehicle in another lane is larger than the speed of the following vehicle.

When the situation satisfies all the above requirements, the car will switch to the aim lane. So far, the normal traffic flow is established.

In the end, there is an accident possibility in the system. We assume that every car in any timestamp has the same accident possibility, and if the accident occurs, it will last randomly 20 timestamps to 50 timestamps because it needs time to clean the road.

3. The program of the simulation model

Considering our group members' advantages and the complexity of the traffic model, we selected C as the language to build this simulation model. So, this part of tutorial will be mainly from perspective of C code. You can use any language you are adept at.

We'll first introduce libraries you need, global variables we defined and the most important-the struct we used. Then, the frame work of the main function will be illustrated to let you have a conceptual sense that how the model work as a whole. In the end, several significant subfunctions will be explained to show you the detailed mechanism of this Cellular Automaton Model.

3.1 Global variables and struct

According to the theoretical model descripted on the previous chapter, there are some variables appear frequently and keep constant all the time, we define them as global variables:

```
1   #define DENSITY 0.2      //The original density of traffic on the road
2   #define P_REDUCE 0.25    //The possibility of car to reduce their speed
3   #define MAX_SPEED 3      //Max speed of each car
4   #define ROAD_LENGTH 100  //The total length of the road
5   #define CELL_LENGTH 1    //The length of each cell
6   #define MAX_SPACE (int)(ROAD_LENGTH / CELL_LENGTH)  //Number of cells
7   #define MAX_TIME 500     //The time of the whole simulation process
8   #define ACCIDENT_PROB 0.000001  //Probability of accident on each car
9   #define MAX_ACCIDENT_PERIOD 50  //The max lasting time of accident
10  #define MIN_ACCIDENT_PERIOD 20  //The min lasting time of accident
```

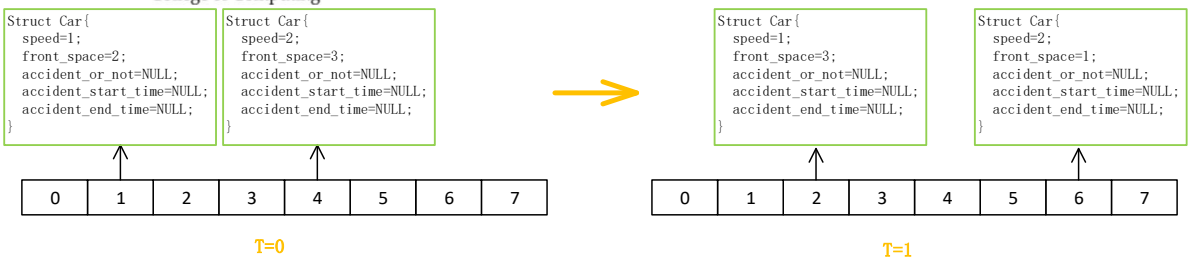
All the variables above are adjustable to achieve different results that could be used to analyze interrelationship among disparate variables or used to suit various realistic situation. However, there are some tips of changing variables:

a. DENSITY is the original density when you randomly generate cars on the road, it's not fixed and will change through time;
b. P_REDUCE is a real statistics data. On the highway it's 0.25 and 0.5 for cars on municipal road;
c. ROAD_LENGTH / CELL_LENGTH should be a constant and a number big enough to get effective observation;
d. ACCIDENT_PROB is extremely sensitive to the simulation result, because this is the accident rate on each car, not the entire traffic flow;

Next, we come to the structs, the major data structure we will operate through the whole program:

```
1  ////////////////////////////////////////////
2  // Define a struct, contains info of one car
3  ////////////////////////////////////////////
4  struct Car
5  {
6      int speed;                //Currrent speed of this car
7      int front_space;          //Number of empty space ahead this car
8      int accident_or_not;      //If accident happened on this car,
9                                  the value is -1
10     int accident_start_time;  //Time when accident starts
11     int accident_end_time;    //Time when accident ends
12 };
13
14 ////////////////////////////////////////////////
15 // Global structs, represent 1st lane to 5th lane
16 ////////////////////////////////////////////////
17 struct Car *lane1[MAX_SPACE];
18 struct Car *lane2[MAX_SPACE];
19 struct Car *lane3[MAX_SPACE];
20 struct Car *lane4[MAX_SPACE];
21 struct Car *lane5[MAX_SPACE];
```

Each "struct Car" represent a car, which has all information you need to run the simulation. We created 5 pointer arrays of "struct Car" type as the 5 lanes of the highway, "cars" will move from laneX[i] to laneX[i+n], as pointers change their objects. Moreover, lane changing would happen like lane1[i] to lane2[i]. The data structure is shown as following:



Cars' movement is realized via changing the index of array element or changing the array it belongs to.

3.2 The main function

The code of the main function is shown below, the only parameter we pass in is the name of a .txt file that stores the data of simulation results:

```
1   int main (int argc, const char * argv[]){
2       FILE* ofp;
3       if ((ofp = fopen(argv[1], "w")) == NULL)  printf("Error! Opening file\n");
4
5       /* initialize the variable */
6       int accident_time[100] = {0};
7
8       /* initialize cars */
9       CarsGenerator();
10      printf("------ time: 0 ------");
11      PrintFiveLanes(ofp, 0);
12
13      int a_index = 0;
14
15      /* start the system */
16      for(int i=1; i<MAX_TIME; i++){
17
18          MoveCars(i);
19          TransferLanes();
20              one_car_accident(i);
21
22          /* test accident time */
23          if(if_accident == 100){
24              if(a_index < 100) accident_time[a_index] = i;
25              if_accident = 0;
26              a_index++;
27          }
28
29          PrintFiveLanes(ofp, i);
30
31      }
32
33      /* print accident time */
34      printf("accident times: \n");
35      for(int i=0; i<100; i++){
```

```
36          printf("%d\n", accident_time[i]);
37      }
38
39      /*free memory*/
40      FreeCars();
41  }
```

For the argument, we chose the filename where we store the result data as the only argument here. All the parameters we defined above could be set as arguments, which depends on your purpose of the simulation. For instance, if you are interested in the interrelationship between accident probability and initial traffic density, you could simply set these two as arguments and keep others as global variables, in this way, you could run multiple experiments without changing the code inside the program.

There are 5 parts of in the main function:

a. Open a .txt file with writable mode, ready to store the simulation data produced in the followed stage;
b. Initialize cars: Randomly generate cars in the road that consists of 5 lanes;
c. Start the system: Take the index i of the FOR loop as timer, each i represents one time step and the system will completely execute once per step: Move cars according to the CA principles, randomly change some cars' lane, randomly produce accident on some cars and keep their speed at 0 until accident ends, record the time point that accidents happened, print out every car's information in every time step both to the screen and .txt file we opened above;
d. Print out accident time: Easy to check while the system run well;
e. Free memory: Free the memory allocated to a huge number of "struct Car".

3.3 Subfunctions

In this part, we will go deeper into the program to see how to convert theoretical model to feasible C program. The basic subfunctions you need to implement the traffic flow vs. accidents simulation are listed as below:

a. Initial traffic flow generator: This function will generate cars on random position on each lane according to the density variable. We assume all original cars are safe that means no accident occurs at the beginning. Besides position, you also need to generate their speed, so they can automatically "move" in following time steps;
b. New car generator: Once the simulation begins, original cars will gradually leave the lanes, so you need a new car generator to consecutively produce new cars and insert them into the head the lanes;

c. Existing car eliminator: When cars are approaching the end of the lane, we need a function first to judge if the car is leaving the lane at the next time step, if yes, then eliminate it from the lane;
d. Move car function: This function will do operations on each car in every time step: accelerate or decelerate according to the space ahead the car, randomly reduce the speed of the car, keep speed of accident car at 0, then change the position by car's speed；
e. Lane transfer function: This function will move car from laneX[i] to laneY[i], if the target cell is empty and car behind this cell is far enough, in only one-time step. In the reality, drivers will change the lane depending on the situation they are in. To make the simulation as real as possible, we must add this function, and therefore we need to build more than one lane.;
f. Accident producing function: This function will first determine if an accident happens on this this car. Once the accident happens, it will generate the lasting time of the accident, so the *Move car function* will keep its speed at 0 during the accident period. In this way, the congestion will occur with the time step increasing.
g. Print function: Print the result data into files or other data carriers. This function is also significant, we need the simulation data to verify if the program runs well and to do analysis with these data.

3.3.1 Initial traffic flow generator

```
1   //generate random numbers within a range without duplication and with
    ascending order
2   int* RandomGenerator(){
3       static int r[(int)(MAX_SPACE*DENSITY)];
4       int arr[MAX_SPACE]={0};
5       for(int i=0; i<(int)(MAX_SPACE*DENSITY); i++){
6           r[i] = rand()%MAX_SPACE;
7           while(arr[r[i]] != 0){
8               r[i] = rand()%MAX_SPACE;
9           }
10          arr[r[i]] = 1;
11      }
12      //sort array
13      for(int i=0; i<(int)(MAX_SPACE*DENSITY); i++){
14          for(int j=i+1; j<(int)(MAX_SPACE*DENSITY); j++){
15              if(r[i]>r[j]){
16                  int temp = r[i];
17                  r[i] = r[j];
18                  r[j] = temp;
19              }
20          }
21      }
22      return r;
```

```
23  }
```

```
1   //generate cars
2   void CarsGenerator(){
3       TimeSeed();
4
5       //lane1
6       int* random_nums = RandomGenerator();
7       int random_nums_index = 0;
8       for(int i=0; i<(int)(MAX_SPACE); i++){
9           // malloc
10          if((lane1[i] = (struct Car*)malloc (sizeof(struct Car))) == NULL)
    {printf("malloc error\n"); exit(1);}
11          // allocate cars
12          if(random_nums_index<(int)MAX_SPACE*DENSITY && i ==
    random_nums[random_nums_index]){
13              lane1[i]->speed = rand()%(MAX_SPEED+1);
14              if(random_nums_index < (int)MAX_SPACE*DENSITY - 1) lane1[i]-
    >front_space = random_nums[random_nums_index+1] -
    random_nums[random_nums_index] - 1;
15              else lane1[i]->front_space = MAX_SPEED+1;
16              random_nums_index++;
17          }else{
18              lane1[i]->speed = -1;
19          }
20      }
21
22      //lane2
23      ......
24
25      //lane3
26      ......
27
28      //lane4
29      ......
30
31      //lane5
32      ......
33  }
```

In this part, the cars are generated and initialized randomly on the lanes, and we assume that the numbers of cars on each lane are equal originally. The random location on each lane is done by generating an array with random numbers in a range without duplication, which is done by the function *RandomGenerator().* Then, for the cells with cars in, the speed is a random number between 0 and 3, which is fixed at the beginning, and for the cells with on cars in, the speed is -1 which is illegal so that we can check if there is a car in the cell.

3.3.2 New car generation and elimination

To simplify our model, we choose the *Jamitons on circular road [Sugiyama et al.: New J. of Physics 2008]* as the prototype of each lane, which means the total number of cars is fixed and determined by the preset density of traffic flow. In another word, the total number of "struct Car" are fixed-the memory space is limited, too. For generation and elimination of cars, the functions should follow the rule that only if a car left the lane, another new car could enter the lane. The whole lane is like a loop shown below:

Notice that, although we use the same "Car structs", whenever a "Car struct" re-enter the lane, its speed should be refreshed to a random number.

3.3.3 Moving cars

```
1   // move cars to the next position
2   void NextPosition(struct Car *lane[], int current_time){
3       int if_new_car = 0; // judge whether it should generate a new car
4
5       for(int i=MAX_SPACE-1; i>=0; i--){
6           if(lane[i]->speed != -1){
7               //Acceleration process
8               lane[i]->speed = lane[i]->speed + 1;
9               if(lane[i]->speed > MAX_SPEED) lane[i]->speed = MAX_SPEED;
10
11              //Deceleration process, if there is not enough space
12              if ((lane[i]->speed) > (lane[i]->front_space)) {
13                  lane[i]->speed = lane[i]->front_space;
14              }
15
16              //Randomization process, reduce speed with prob p
17              if (lane[i]->speed > 0) {
```

```
18                  double prob_decelerate = (double)rand() / RAND_MAX;
19                  if (prob_decelerate <= P_REDUCE) lane[i]->speed = lane[i]-
    >speed-1;
20              }
21
22              //Keep speed of accident car to 0
23              hold_accidental_car(lane, current_time, i);
24
25              //Move this car to new position, Xi -> Xi + Vi
26              int p = i + lane[i]->speed;
27              if(p >= MAX_SPACE){
28                  if_new_car = i;
29                  continue;
30              }
31              if(lane[p]->speed == -1){
32                  lane[p]->speed = lane[i]->speed;
33                  lane[i]->speed = -1;
34              }
35          }
36      }
37
38      if(if_new_car != 0){
39          int p = NewCarGenerator(lane);
40          lane[p]->speed = lane[if_new_car]->speed;
41          lane[if_new_car]->speed = -1;
42      }
43      UpdateFrontSpace(lane);
44  }
```

This function should follow these steps in every time step:

Step 1. All the vehicles whose velocity has not reached the maximum $vmax$ will accelerate by one unit;
Step 2. Assuming a car has m empty cells in front of it. If the velocity of the car (v) is bigger than m, then the velocity becomes tom. If the velocity of the car (v) is smaller than m, then the velocity changes to v. ($v \rightarrow min[v, m]$);
Step 3. The velocity of the car may reduce by one unit with the probability p;
Step 4. If a car is in accident status, change it to 0 (In Step 1, 2, 3, the speed of accident car will be changed) if current time step is smaller than the end time of the accident;
Step 5. After 4 steps, the new position of the vehicle can be determined by the current velocity and current position. ($xn' \rightarrow xn + vn$).

3.3.4 Lane transfer

```
1   // check whether need to transfer the lane
2   void CheckTransfer(struct Car* l1[], struct Car* l2[], int* transfer_cars){
3       for(int i=0; i<MAX_SPACE; i++){
```

```
4           if(l1[i]->speed != -1){
5               // 1 The distance ahead in current lane is smaller than the car
    speed.
6               if(l1[i]->front_space < l1[i]->speed){
7                   // 2 The distance ahead in another lane is larger than in the
    current lane.
8                   int j=0;
9                   while(i+j+1<MAX_SPACE && l2[i+j+1]->speed == -1){
10                      j++;
11                  }
12                  if(i+j+1>=MAX_SPACE || j > l1[i]->front_space){
13                      // 3 There exists an empty cell right in another lane.
14                      if(l2[i]->speed == -1){
15                          // 4 The distance ahead of the following vehicle in
    another lane is larger than the speed of the following vehicle.
16                          j=0;
17                          while(i-j-1>=0 && l2[i-j-1]->speed == -1) j++;
18                          if(i-j-1<0 || l2[i-j-1]->speed <= j){
19                              // the car should transfer the lane
20                              transfer_cars[i] = 1;
21                          }
22                      }
23                  }
24              }
25          }else{
26              transfer_cars[i] = 0;
27          }
28      }
29  }
30
31  // if the same car can transfer to both the right lane and the left lane
32  // we assume that it has 1/2 possibility choosing the right lane
33  // and 1/2 possibility to choose the left lane.
34  void SolveDuplication(int* a, int* b){
35      if(a == NULL || b == NULL){printf("error in solving duplication!\n");
    exit(1);}
36      for(int i=0; i<MAX_SPACE; i++){
37          if(a[i] == 1 && b[i] == 1){
38              int p = rand()%100;
39              if(p<50) b[i] = 0;
40              else a[i] = 0;
41          }
42      }
43  }
44
45  //transfer the cars
46  void TransferOneLane(struct Car* l1[], struct Car* l2[], int* transfer_cars){
47      //change status
48      for(int i=0; i<MAX_SPACE; i++){
49          if(l1[i]->speed!=-1 && l2[i]->speed==-1 && transfer_cars[i] == 1){
50              l2[i]->speed = l1[i]->speed;
51              l1[i]->speed = -1;
52          }
53      }
```

```
54  }
```

Lane transfer also has some prerequisites, check all these conditions before execute lane transfer for each car, as *Chapter 2* described:

Step 1. The distance ahead in current lane is smaller than the car speed;
Step 2. The distance ahead in another lane is larger than in the current lane;
Step 3. There exists an empty cell right in another lane;
Step 4. The distance ahead of the following vehicle in another lane is larger than the speed of the following vehicle.

Notice that we need to have two kinds of lane-changing operation. Imagine our road like this: 1-2-3-4-5, each number represents a line. Cars on edge lanes-lane 1 and lane 5, cars only have one direction to change their lane, meanwhile, cars on lane 2, 3, 4 have two direction to change their lane. When it comes to lane 2, 3, 4, we need a mechanism for cars decide which direction they will transfer to, left or right.

Further, for lane 2,3,4, they have one lane on the left and one lane on the right, we need a mechanism to prevent conflicts between lane-changing cars. For instance, car A is on lane 2, car B is on lane 4, they are driving parallelly. When they both want to change into lane 3, we need to keep one car in its current lane and let another one change lane first.

3.3.5 Accident generation

```
1   //Randomly generator a accident lasting time
2   int accident_period_generator() {
3       int accident_lasting = (int)rand() / (MAX_ACCIDENT_PERIOD -
    MIN_ACCIDENT_PERIOD) + MIN_ACCIDENT_PERIOD;
4       return accident_lasting;
5   }
6
7   //produce accident for possible car, if accident happen in this function,
    return the accident lasting period
8   void one_car_accident_generator(struct Car* lane_accident[], int current_time,
    int accident_period) {
9
10      //decide if accident happen on this car
11      for (int i = 0; i < MAX_SPACE; i++)
12      {
13              double prob = (double)rand() / RAND_MAX;
14              if (lane_accident[i] != NULL && lane_accident[i]-
    >accident_or_not != -1)
15              {
16                      if (prob<=ACCIDENT_PROB)
17                      {
18                  if_accident = 100;
```

```
19                                  lane_accident[i]->accident_or_not = -1;
20                      lane_accident[i]->speed=0;
21                                  lane_accident[i]->accident_start_time =
    current_time;
22                                  lane_accident[i]->accident_end_time =
    accident_period + current_time;
23                              }
24                  }
25          }
26  }
```

To simplify the model, we assume that the accident happens only on one car each time, which is to say, no accident will influence more than 1 car. Our focus is the impact of congestion to the whole traffic flow, thus the number of cars that involved in the accident is not so important for the system here.

In every time step, the function will generate a relatively small probability that an accident will happen on a car. Then check all cars one by one if some of them are unlucky enough to get the accident. Next, in accident cars' structs, the accident index *accident_or_not* will changed to -1, and they will get a *accident_start_time* and *accident_end_time* in their "Car struct". Finally, in the *Moving cars* function, after acceleration / deceleration process, one subfunction will check the value of *accident_or_not* of each car, if it's -1 and the value of time step (index i of the FOR loop in the main function) is between *accident_start_time* and *accident_end_time* (Accident is not ended yet), the speed will be changed to 0 (Speed of accident car will remain 0 and stay in the same position until the accident ends). In this way, cars behind will be forced to decelerate until 0 or change their lane-congestion formed.

3.3.6 Data print

```
1   //print cars
2   // format:
3   // time, lane#, i-th, speed, location, front space
4   void PrintCarList(FILE* ofp, struct Car *lane[], int timestamp, int lane_id){
5       if(ofp == NULL) {printf("print error: car_list is NULL\n"); exit(1);}
6
7       int id = 0;
8       for(int i=0; i<MAX_SPACE; i++){
9           if(lane[i]->speed == -1) continue;
10          else{
11              fprintf(ofp, "%d %d %d %d %d %d\n", timestamp, lane_id, id,
    lane[i]->speed, i, lane[i]->front_space);
12              printf("%d %d %d %d %d %d\n", timestamp, lane_id, id, lane[i]-
    >speed, i, lane[i]->front_space);
13              id++;
```

```
14              }
15          }
16      }
```

The format of the printed data depends on how your data analysis program will read them, you can print the data in the way you like. Sample program is shown above.

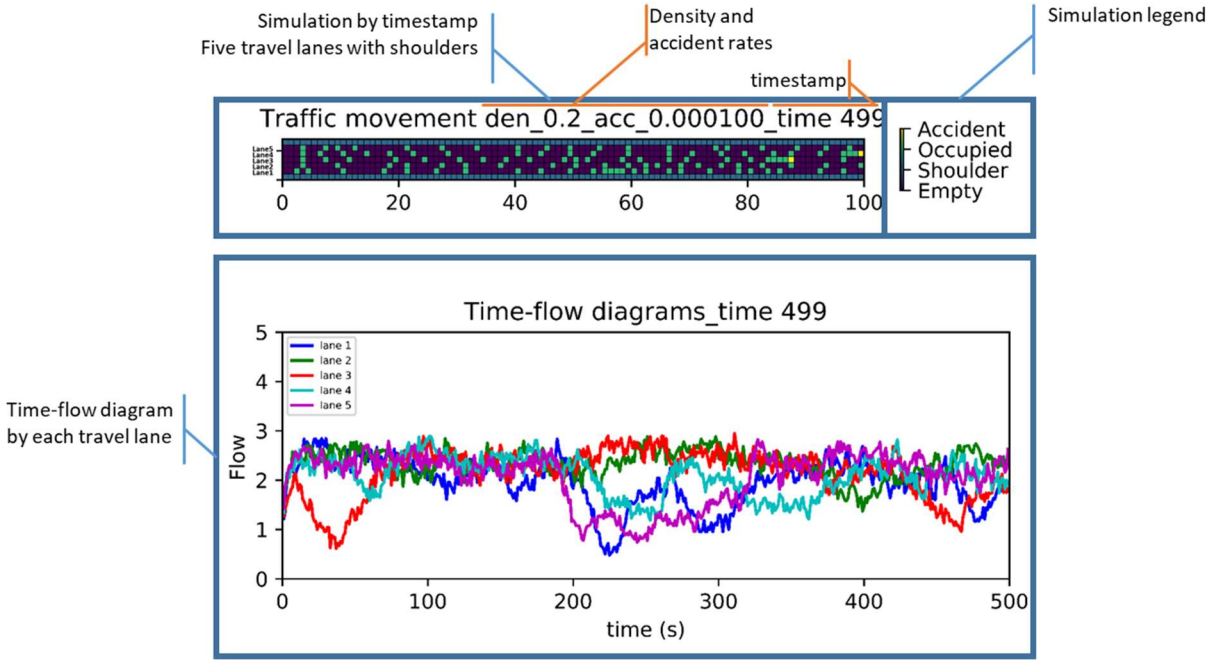
4 Data visualization and results analysis

While selecting C as programming language to build this simulation model in the previous section, we selected python 2.7 in order to express visualized simulation with features discussed in the previous sections of how to move vehicles on 5 travel lanes with accidents.

In this section, we will introduce how to crease data visualization and discuss the results based on den_0.2_acc_0.000100.txt and den_0.2_acc_0.000100_accident.txt randomly selected as an example.

4.1 Simulation output diagram

Two main parts are presented in the simulation output diagrams: simulation and time-flow diagram.

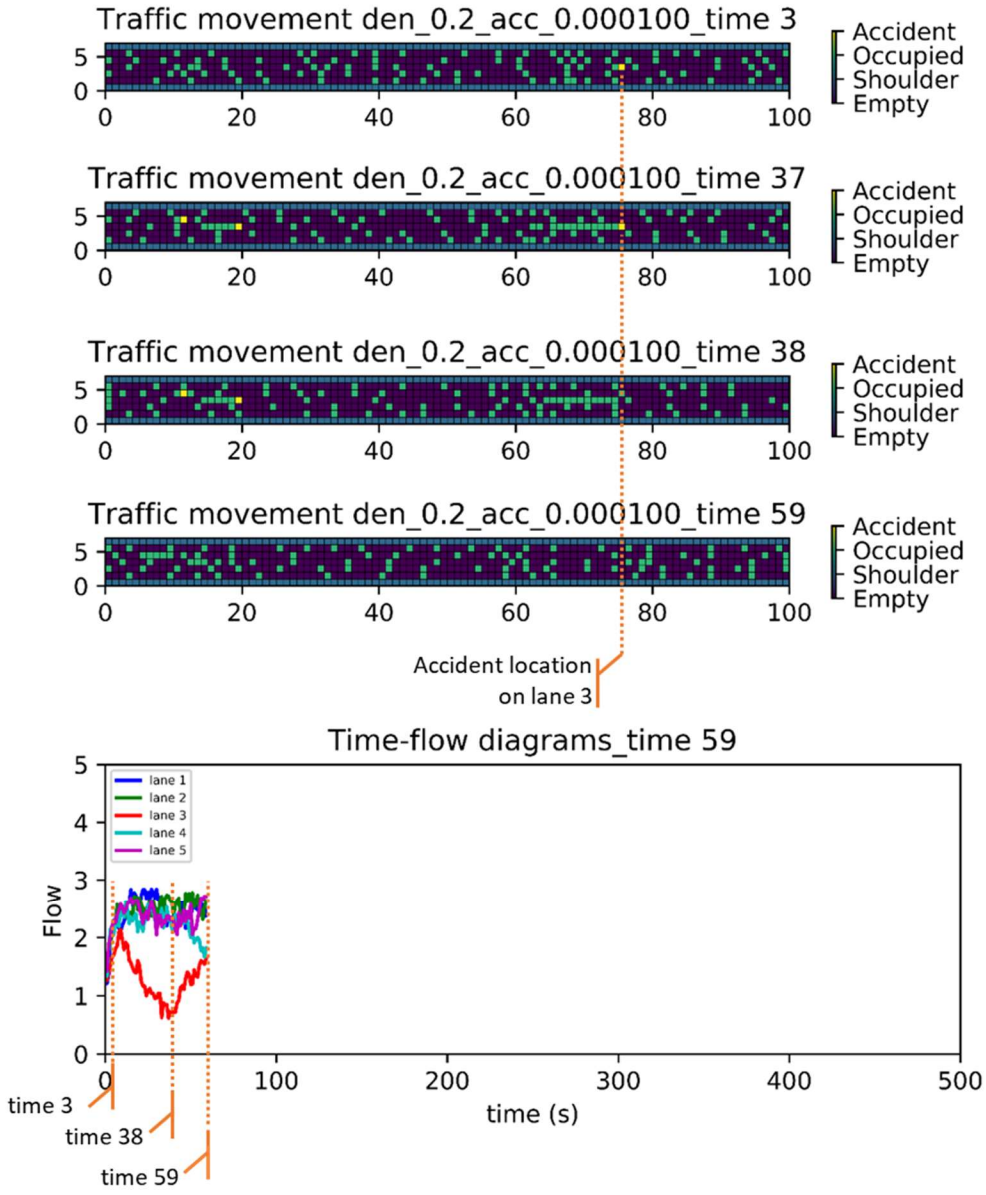


This dynamic process could be seen at YouTube:
https://www.youtube.com/watch?v=yssmOvoTccU

4.2 Results

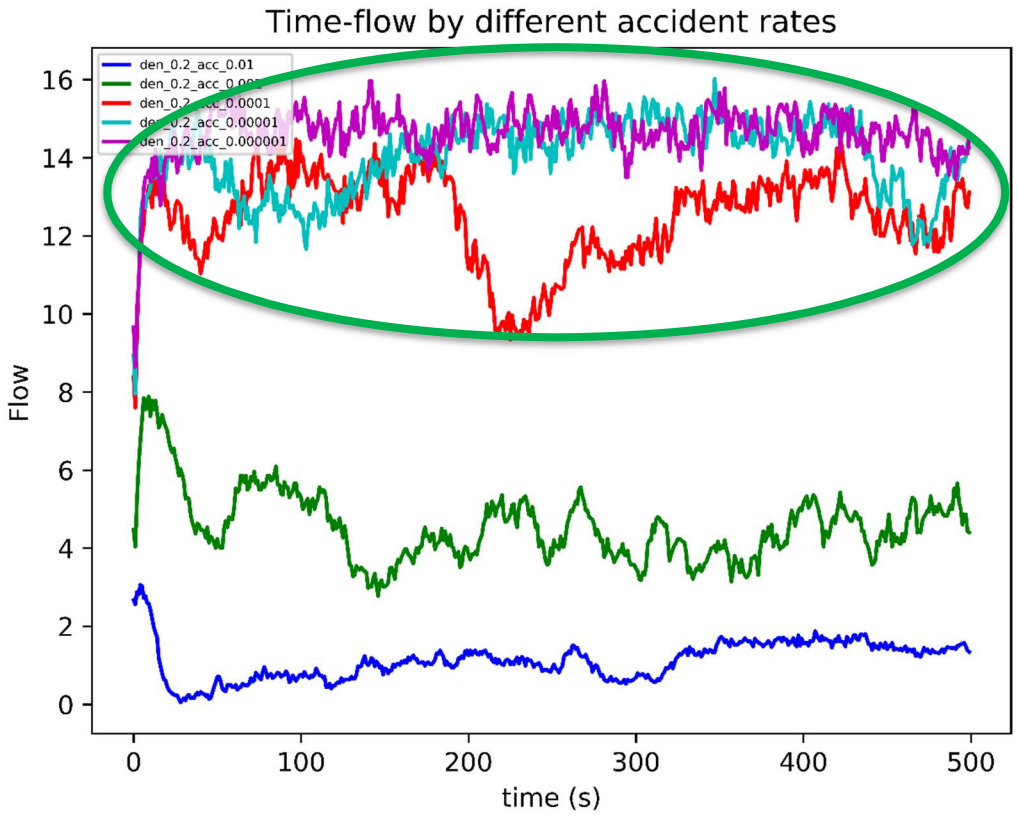
The following are three screenshots related to the first accident within density 0.2 and accident 0.0001. The figure below is when the accident happened at time 3 and resolved at time 38. We can see the yellow dot which represent an accident in the simulation along with orange dotted line noting of 'accident location of lane 3'. In the second simulation, it is when the accident is about to clean which you can see the yellow dot in that location at the last in the simulation and in time 38 you cannot see the yellow dot in that location any more. In the third simulation in the figure is just after the accident was clear, but it has still congestion queue backed up downstream of travel lane and from the accident location the congestion starts to remove backward. In the fourth simulation in the figure below the congestion due to the accident completely cleans up.

In the diagram of time-flow chart in the figure below, we can see the how the flow fluctuates by time on each lane (lanes are colored coded in the legend) When the accident happens, the flow is not affected by the accident. At some point of time between starting point of timestamp and ending point of timestamp, the flow peaked high and then until time 38, the flow goes down due to congestion. Also, at time 38, when the accident scene is cleaned up, the flow starts to go up. Over 500 seconds, we can see these patterns as many as accidents happens in each lane.


Traffic movement den_0.2_acc_0.000100_time 3
Traffic movement den_0.2_acc_0.000100_time 37
Traffic movement den_0.2_acc_0.000100_time 38
Traffic movement den_0.2_acc_0.000100_time 59
Accident
Occupied
Shoulder
Empty
Accident location
on lane 3
Time-flow diagrams_time 59
lane 1
lane 2
lane 3
lane 4
lane 5
Flow
time (s)
time 3
time 38
time 59

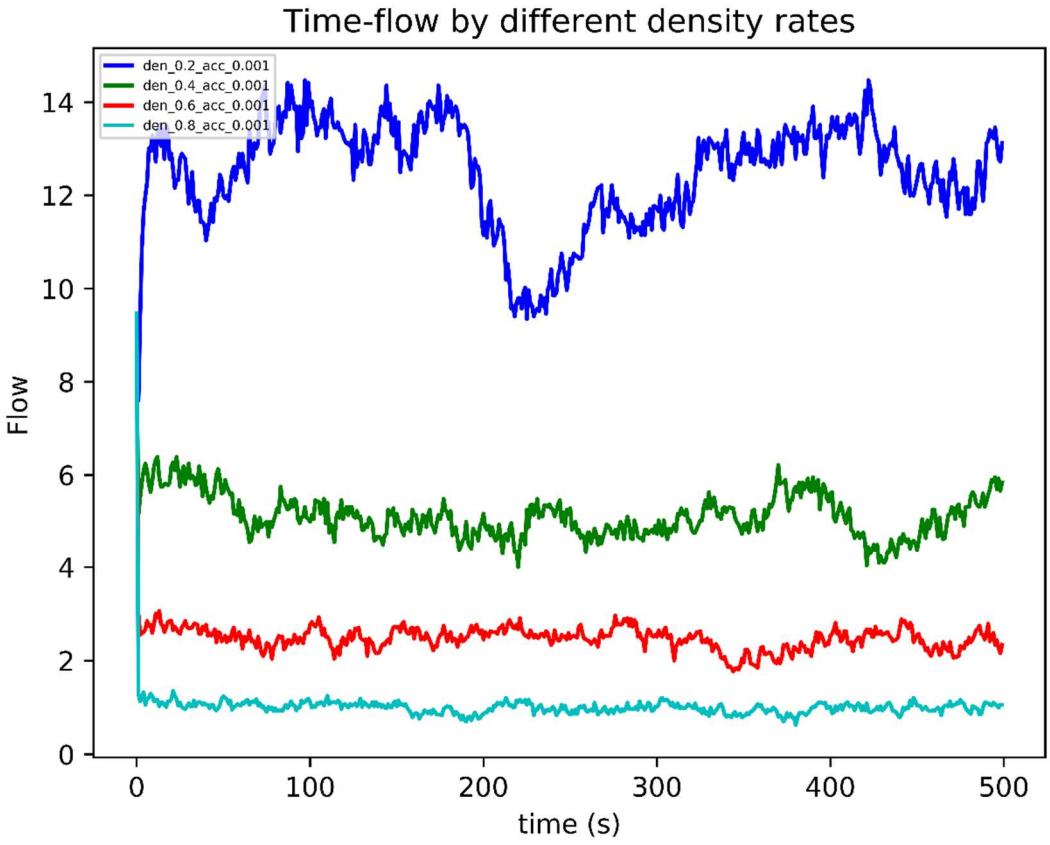
4.3 Time-flow chart by density and accident rates

In this section, we discuss how the flow generates by density or accident rates. We selected density 0.2 with different accidents rate such as 0.01, 0.001, 0.0001, 0.00001, and 0.000001.



As seen in the figure above, even though each time-flow line is set up with different accident rates, all lines are fluctuated by the chance of accident each time. Generally, when accident rates are high, the flow decrease. When the accident rates are below than 0.001, the flow lines are located pretty much in the similar flow ranges. Comparing to other two cases (acc_0.01 and acc_0.001), those three time-flow lines (in green circle) with lower accident rates have higher flow rates in all times.

Similarly, when holding the accident rate at 0.001, the time-flow lines are presented in the following chart with different density rates such as 0.2, 0.4, 0.6, and 0.8. In the traffic flow theory, all speed, flow, and density are related to each other, and because flow is the product of speed and density, the relationship between flow and density is inverse. As seen in the figure, when the density rates get increased, generally the average of flow decreases. Unlikely to the green circle in the previous figure, the time-flow lines clearly are different to each other.


Time-flow by different density rates
den_0.2_acc_0.001
den_0.4_acc_0.001
den_0.6_acc_0.001
den_0.8_acc_0.001
Flow
time (s)
0
2
4
6
8
10
12
14
0
100
200
300
400
500

Georgia Tech School of Computer Science
College of Computing

Reference

[1] AJC, “Atlanta traffic among worst in the world, study finds” http://www.ajc.com/news/local/atlanta-traffic-among-worst-the-world-study-finds/C6JR110E1z9xZeGGmjJ2HM/, accessed 2/10/2019

[2] K. Nagel, M. Schreckenberg (1992). A cellular automaton model for freeway traffic, Journal de Physique I 2(12), 2221-2229.

[3] Newell G.F. (2002) A simplified car-following theory: a lower order model, Institute of Transportation Studies, University of California, Berkeley.

[4] M. Kanai, K. Nishinari, T. Tokihiro (2006) Stochastic Cellular-Automaton Model for Traffic Flow, International Conference on Cellular Automata, pp 538-547

Work allocation

Ruxu Zhang：

1. Coded for the simulation system building, lane generation, lane switching and output management;
2. Helped on the visualization part by coding a flow calculator;
3. Wrote the introduction and model part in tutorial.

Ciyuan Yu：

1. Planned the system structure and data structure of the simulation;
2. Coded the simulation system, car moving functions, accident handling functions;
3. Wrote the proposal, check point 1, and chapter 3 of the tutorial.

Han Gyol Kim：

1. Literature searching and review;
2. Coded for the data visualization and analysis program, analyzed the data of the simulation results;
3. Wrote the proposal, check point 1, and chapter 4 of the tutorial.