Traffic Flow: Peachtree City Simulation

Dawar Ahmed, Robert Hill Pendley

Code: https://github.gatech.edu/dahmed7/CX4230-Project-2

Cellular Automata

Problem Statement:

We are going to be investigating the average travel time for vehicles to traverse a portion of Peachtree Street, the corridor from 10th street to 14th street. We will focus on differing various conditions to find the distribution of travel times through the corridor. We will implement random lanes closures (due to car's breaking down with random probability p) and investigating how this will affect the travel time. We also be adding random probability that the driver is a rash driver which is affect their probability to change lanes, thus creating ripples behind the car and he moves in and out of lanes, affecting travel times.

Conceptual Model:

This part of the project will be using cellular automata to model and simulate the project. This means our world consists of a world of open cells in a grid which can either be occupied (meaning a car is present) or unoccupied (meaning a car is not present). Initially the grid is filled with N% of cars where N is the starting density. This density is specified by the specific simulation and can and will be changed throughout our project to see how this will affect how travel time. Our model is based in a simulated world which consists of a 2 x 135 Numpy array which will use as our backing for our cellular grid. The 2 represents the number of lanes on Peachtree Street and 135 represents the physical distance from 10th Street and Peachtree up to 14th. This number is the actual physical distance divided by the average length of car, meaning there are 135 car sized cells from 10th street to 14th street. We have also placed two traffic lights in the model, one 40 cells up and the other 80 cells up. These are just approximately placed as they seen on Peachtree Street. Once we start the simulation, either two or one cars will be generated per unit time based on the specific simulation and the traffic density in that specific simulation. Once the simulation begins the cars propagate forward with a given set of rules which we will discuss later. The cars also have the ability to change lanes and there is a random probability that a car breaks down which has initially been defined as 1/500. Cars may only the max velocity which we have defined as 3. Once we can run the simulation for a period of time we are able to determine data on how long each car stays in the system. In the Final Project we will be using this data to validate and verify our results.

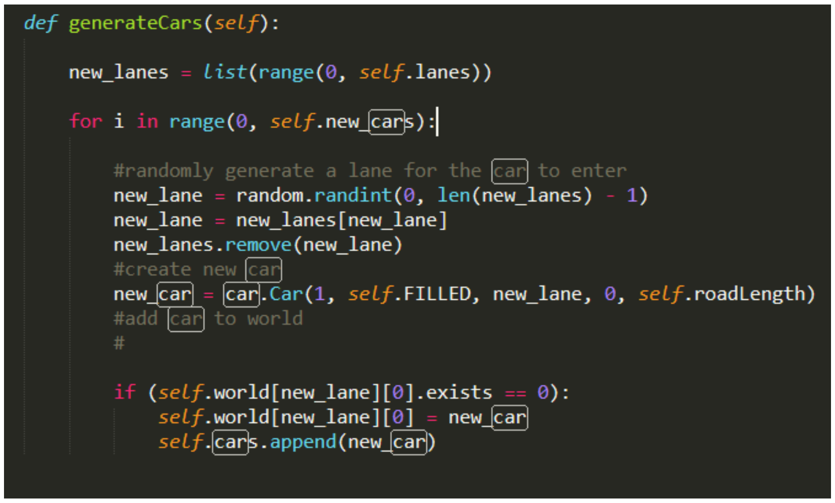
Our model simplifies the changing lanes by only having the car move completely sideways as opposed to the more diagonal motion seen in real life. Also, our model assumes the area outside the visible world is empty. As in beyond the grid there are no cars and no speed restrictions which ultimately affects the simulation especially as you approach the end of the grid since the car may speed up even in high density traffic situation. We have also, for the time being, grossly over simplified the two traffic lights. Currently the traffic lights only exist as either on or off (green or red) and each state is for 10 seconds. So, the simulation using a timer, switches between the green or red every ten seconds. In the real world you would also have amber in the traffic light and there is a set of completely different car actions which happen in that situation. Also, our traffic lights are arbitrarily offset by three seconds just to make sure both weren't stopping and starting at the same time.

Our model assumes that each car is travelling straight and is only travelling straight. So, each car should have no intention of switching lanes (in order to turn) since its sole object is to move forward as quickly as possible. We also assume that there will be no car accidents, which is certainly not true on real life roads. Once a car breaks down, it doesn't move off the road or onto the side, it is assumed that the car will remain on the road and be fixed in a matter of time (5 units to be precise).

The major limitation we have faced with our model so far is solving the issue of making the cars move in a natural way. Because of the world being limited to just what can be seen, there is a sense of the cars always following a very predictable behavior. The cars will start slow, increase their speed and eventually start moving at max velocity as they approach the end and rarely change lanes. I believe this is because of the free world at the end. So, cars have the ability to increase their velocity at the end of the road, meaning cars before can also increase velocity and thus there is no need for a car to ever change lanes. Another issue is the appearance of a convoy. Cars will at times move in tight packs with almost no space in between, which I don't believe is reflective of a real road. Cars in the real life would slowly spread out over time.

Specific Code Implementations:

1. Generating Cars. Here the method generates new cars, where *new_cars* is the number of cars to add each iteration



```
def generateCars(self):

    new_lanes = list(range(0, self.lanes))

    for i in range(0, self.new_cars):

        #randomly generate a lane for the car to enter
        new_lane = random.randint(0, len(new_lanes) - 1)
        new_lane = new_lanes[new_lane]
        new_lanes.remove(new_lane)
        #create new car
        new_car = car.Car(1, self.FILLED, new_lane, 0, self.roadLength)
        #add car to world
        #

        if (self.world[new_lane][0].exists == 0):
            self.world[new_lane][0] = new_car
            self.cars.append(new_car)
```

2. Propagate Forward. Here is the method where the cars move forward, and other elements are updated in each unit time. The code for this method can be found in the propogateForward method in world.py.
    a. First Traffic Light times are updated
    b. The times of blocked cars are updated and are unblocked is the time has elapsed
    c. Next cars may break time (become blocked cars)
    d. Then the cars follow the propagation rules which are as following:
        i. If car velocity is below max and gap until next car allows speeding up, increase speed by 1

        ii. If car velocity cannot be maintained because of the gap until next car is too small, look to change lanes
        iii. If gap until next car is too small and you cannot change lanes, reduce velocity to match the gap
    e. Remove any cars which have moved off the world
    f. Time is incremented for the data plotting purposes
3. Changing Lanes. Before a car can change lanes, it must follow a specific set of checks. When a car wants to change lanes, it investigates all possibly lane changes (left and right). In our case it would only ever check left when right and right when left, but this method can be further applied to a large road with 3+ lanes. These are rules if checks for:
    a. Check if a car exists in the potential new lane. If so, it cannot change
    b. Find the distance to previous car in the new lane and the next car in the new lane
    c. If there is no car behind and the current velocity can be maintained in the new lane based on the distance to the next car, then swap lanes
    d. If the previous car can maintain its velocity and the car can maintain its velocity in the new lanes, then swap lanes

   This method is in the changeLanes function in world.py

All other methods aren't really part of the model but can be viewed on the GitHub repository.

Event-Oriented Queueing Model

Problem Statement:

The goal for the project is to simulate traffic in the stretch of Peachtree Street in between 10th and 14th street. The data to be analyzed is the average travel time for all vehicles depending on certain parameters. These parameters include stoplight greenlight length, stoplight greenlight frequency, and frequency of cars entering the system.

Conceptual Model:

The event-oriented queueing model contains separate software modules for the queueing network model (engine1.c) and the simulation logic(roadway.c). The queueing model contains the software needed to schedule events, run the simulation, and handle the list of future events. We used the necessary software from the example airport simulation engine that was provided to us by Dr. Fugimoto. Using this engine, we developed methods to implement a queueing model for the road to be simulated.

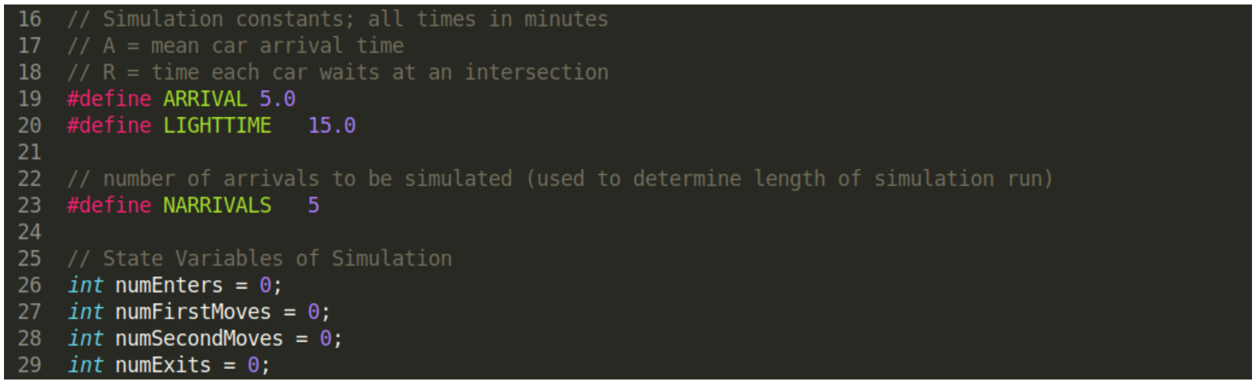
1) Assumptions made
    a. All cars are of the same speed and size
    b. Two cars cannot enter the simulation roadway at the same time.
    c. Cars only enter the roadway at 10th street. (only for checkpoint)

d. The car “enters” the simulation as soon as it crosses 10th street or turns onto Peachtree from 10th street.
e. The car “exits” the simulation as soon as it reaches 14th street
f. Each light takes the same amount of time to pass though (only for first checkpoint)
g. Assume that cars will not be affected by swerves in the road
h. A car cannot exit the simulation at any time
i.

2) Details Concerning the model
   a. The number of cars to be simulated is the constant NARRIVALS
   b. The rate at which cars enter the simulation is determined by an exponential distribution with average ARRIVAL
   c. Each car spends the same amount of time at each stoplight, LIGHTTIME (only for checkpoint)
3) Simplifications made
   a. Cars do not physically “move” distances. They simply queue at fixed points until they can move to the next point. The goal of the simulation is to develop methods and algorithms to determine when the cars are to “move” so that the simulation imitates real life.

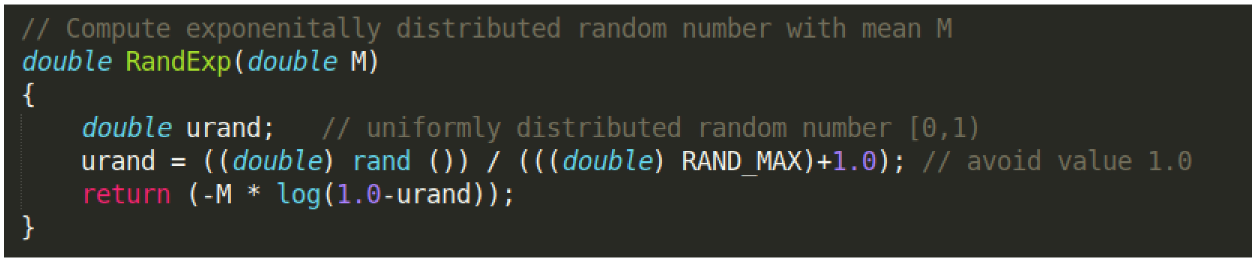
Code Implementations:

1) Important Numbers
   a. ARRIVAL is the mean time between each car being generated
   b. LIGHTTIME is the time each car sits at a light (will be modified after checkpoint)
   c. numEnters, numFirstMoves, numSecondMoves, and numExits is the number of times each event is processed



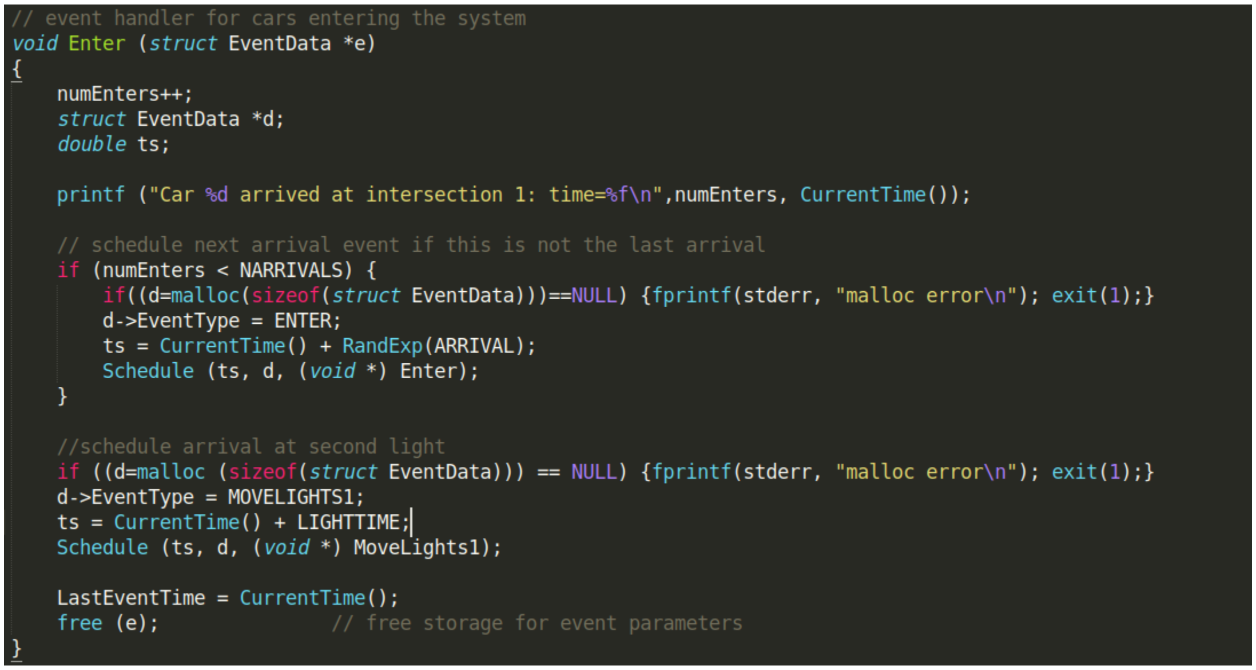
```
// Simulation constants; all times in minutes
// A = mean car arrival time
// R = time each car waits at an intersection
#define ARRIVAL 5.0
#define LIGHTTIME   15.0

// number of arrivals to be simulated (used to determine length of simulation run)
#define NARRIVALS   5

// State Variables of Simulation
int numEnters = 0;
int numFirstMoves = 0;
int numSecondMoves = 0;
int numExits = 0;
```

2) Random number calculation
    a. The second line of the method generates a random number, then divides it by the max random number + 1 to obtain a number between 0 and 1. Then log generates an exponential distribution with mean -1, and we multiply by -M to obtain a number with mean M. This code was obtained from Dr. Fujimoto's example project.



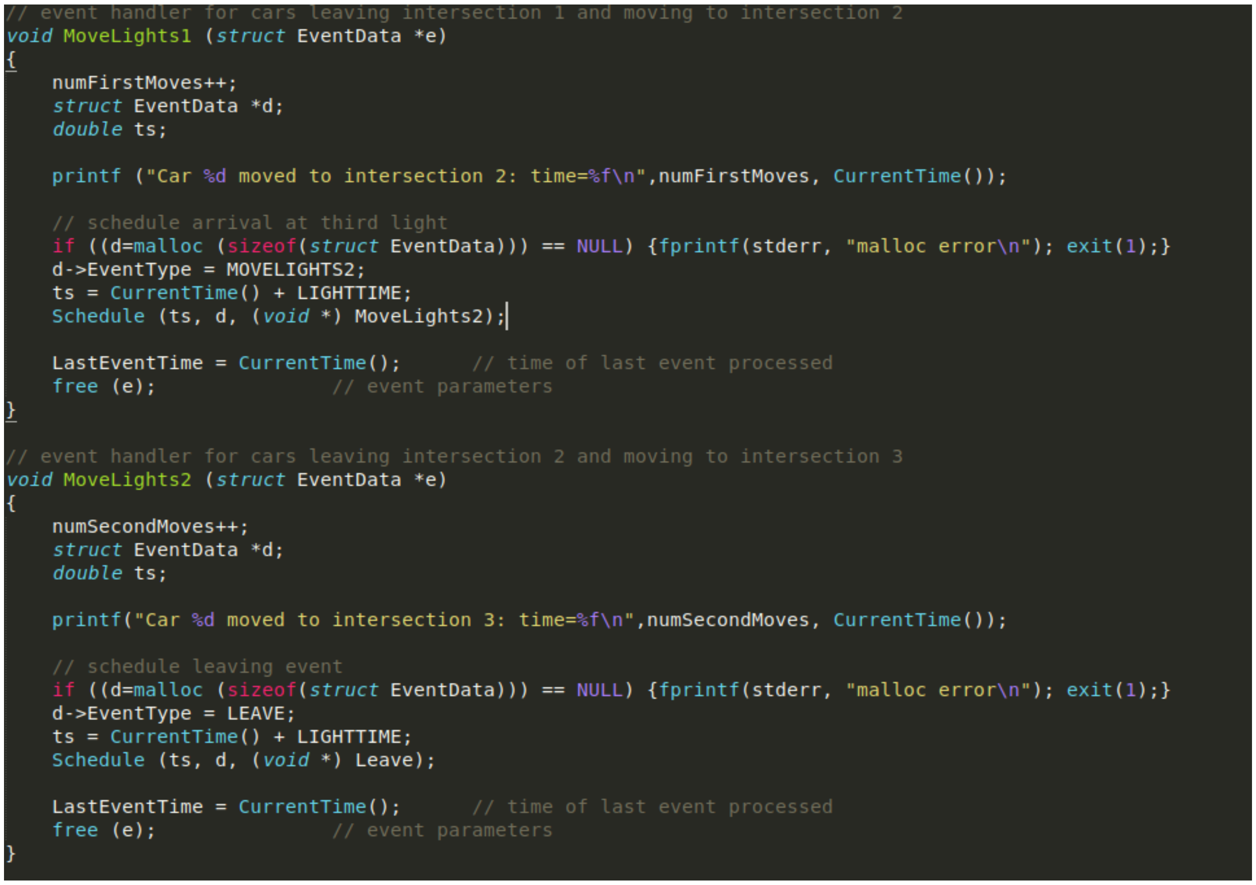
```
// Compute exponenitally distributed random number with mean M
double RandExp(double M)
{
    double urand;   // uniformly distributed random number [0,1)
    urand = ((double) rand ()) / (((double) RAND_MAX)+1.0); // avoid value 1.0
    return (-M * log(1.0-urand));
}
```

3) Entering the system
    a. First the number of enters is increased
    b. Then if there are more cars to be generated, another arrival is scheduled.
    c. Then we schedule the arrived car at the second light
    d. Then we update the time and free the memory pointed to by the parameter



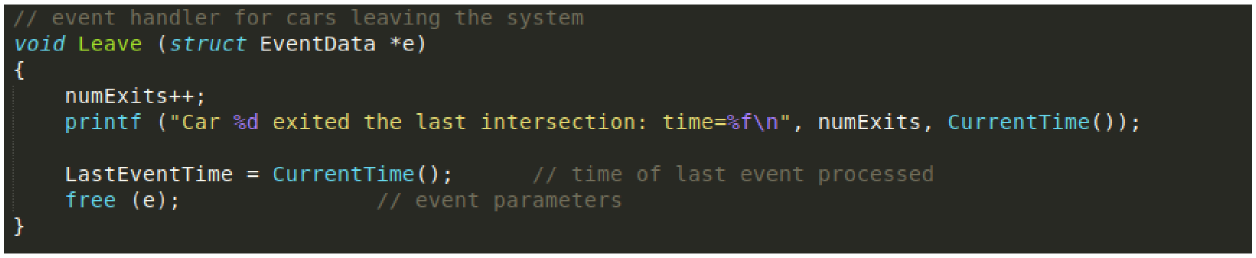
```
// event handler for cars entering the system
void Enter (struct EventData *e)
{
    numEnters++;
    struct EventData *d;
    double ts;

    printf ("Car %d arrived at intersection 1: time=%f\n",numEnters, CurrentTime());

    // schedule next arrival event if this is not the last arrival
    if (numEnters < NARRIVALS) {
        if((d=malloc(sizeof(struct EventData)))==NULL) {fprintf(stderr, "malloc error\n"); exit(1);}
        d->EventType = ENTER;
        ts = CurrentTime() + RandExp(ARRIVAL);
        Schedule (ts, d, (void *) Enter);
    }

    //schedule arrival at second light
    if ((d=malloc (sizeof(struct EventData))) == NULL) {fprintf(stderr, "malloc error\n"); exit(1);}
    d->EventType = MOVELIGHTS1;
    ts = CurrentTime() + LIGHTTIME;
    Schedule (ts, d, (void *) MoveLights1);

    LastEventTime = CurrentTime();
    free (e);                // free storage for event parameters
}
```

4) Moving Intersections
   a. These two functions move the car from intersection 1 to intersection 2, then from intersection 2 to intersection 3.
   b. First increase the number of moves
   c. Then schedule the next event. Either moving or exiting the simulation
   d. Then update the time and free the memory pointed to by the parameter



```c
// event handler for cars leaving intersection 1 and moving to intersection 2
void MoveLights1 (struct EventData *e)
{
    numFirstMoves++;
    struct EventData *d;
    double ts;

    printf ("Car %d moved to intersection 2: time=%f\n",numFirstMoves, CurrentTime());

    // schedule arrival at third light
    if ((d=malloc (sizeof(struct EventData))) == NULL) {fprintf(stderr, "malloc error\n"); exit(1);}
    d->EventType = MOVELIGHTS2;
    ts = CurrentTime() + LIGHTTIME;
    Schedule (ts, d, (void *) MoveLights2);

    LastEventTime = CurrentTime();      // time of last event processed
    free (e);               // event parameters
}

// event handler for cars leaving intersection 2 and moving to intersection 3
void MoveLights2 (struct EventData *e)
{
    numSecondMoves++;
    struct EventData *d;
    double ts;

    printf("Car %d moved to intersection 3: time=%f\n",numSecondMoves, CurrentTime());

    // schedule leaving event
    if ((d=malloc (sizeof(struct EventData))) == NULL) {fprintf(stderr, "malloc error\n"); exit(1);}
    d->EventType = LEAVE;
    ts = CurrentTime() + LIGHTTIME;
    Schedule (ts, d, (void *) Leave);

    LastEventTime = CurrentTime();      // time of last event processed
    free (e);               // event parameters
}
```

5) Leaving the simulation
   a. First increase the number of exit events
   b. Then update the time and free the memory pointed to by the method parameters



```c
// event handler for cars leaving the system
void Leave (struct EventData *e)
{
    numExits++;
    printf ("Car %d exited the last intersection: time=%f\n", numExits, CurrentTime());

    LastEventTime = CurrentTime();      // time of last event processed
    free (e);               // event parameters
}
```