Journal Report 4
9/23/19 - 9/30/19
Richard Zhan
Computer Systems Research Lab
Period 2, White

Daily Log

Monday September 23

I changed the variable $graphCars$ from a vector of Cars to an unordered_map since I want to efficiently remove them after they have been arrived at their destination. I also implemented the method $spawnCar$ for the class $Vertex$. Each Vertex has an assigned $carSpawnRate$, and they will randomly spawn one Car each $1/carSpawnRate$ seconds.

Tuesday September 24

In the method $step\_simulation$, I wrote the code for initializing the newly spawned Cars. On each Car's first turn, it ran A* to determine its route before it started traveling down the first road. I also counted the number of Cars on each Edge and then determined each Edge's speed. The speed of the Cars linearly decreased based on the number of other Cars on the same Edge.

Thursday September 26

I finished the method $step\_simulation$. After determining each Car's speed, I updated its position on the graph. I also added a check to see if the Car had reached its destination Vertex. If so, it would be taken out of the variable 'graphCars' and the Car's trip duration was recorded. For the Car, $Edge$, and $Vertex$ classes, I turned all of their class variables (that were Car/Edge/Vertex objects) into pointers. This allowed the equals operator to behave correctly.

Timeline

Date	Goal	Met
9/9/19 - 9/15/19	Finish coding the basic A* navigation system and collect data on the average amount of time for each trip.	No, I have coded the non-DTD navigation system, but have not written the necessary simulation code to collect data.
9/16/19 - 9/22/19	Finish writing the simulation code and collect data on the average amount of time for each trip.	No, I corrected a bug in the navigation methods for non-DTD cars. I started but have not finished the simulation code.
9/23/19 - 9/29/19	Finish writing the simulation code and tweak variables to reach realistic settings.	Yes, I wrote the necessary code and found the correct input values to run a realistic simulation.
9/30/19 - 10/6/19	Began coding the naive (non-optimized) DTD scheme. Try to finish setting up the class *Event* and the communication system between cars	
10/7/19 - 10/13/19	Finish the naive DTD scheme and begin looking into optimizations	

Reflection

This week, I finished the simulation code for non-DTD cars. I am happy with what I accomplished this week since I was able to meet my goal. My program now runs as expected and the output data supports this. When I run the program with high volumes of Cars, the average trip time increases significantly. I have established the baseline times of the non-DTD navigation system for my proof-of-concept scheme. I have also scaled the input variables, so that the program runs in a realistic SI unit scale. I am looking forward to the next couple of week since up to this point, much of the coding has been the simple, but tedious work. I am really interested in seeing how faster I can make the Car-to-Car communication systems compared to the naive implementation where each Car is checked against the other Cars $O(N^2)$. Farther down the road, I also want to try making my code run in parallel.

The below code is my method *step_simulation* which performs the bulk of the simulation.

```
/**
 * Spawns new cars
 * Updates car positions
 * Checks if cars have arrived at destination
 */
void step_simulation() {
  // cout << "STEP " << CURRENT_TIME << endl;
  // cout << "CURRENT_CAR_COUNT " << CURRENT_CAR_COUNT << endl;
  // cout << "TOTAL_CAR_COUNT " << TOTAL_CAR_COUNT << endl;


  // Spawn cars
```

```
    for (int i = 0; i < graphVertices.size(); i++) {
      graphVertices[i].carCount += TIME_STEP * graphVertices[i].carSpawnRate;
      while (randDouble(0.0, 1.0) < graphVertices[i].carCount) {
        // cout << "Creating car " << i << " " << graphVertices[i].carCount << endl;
        graphVertices[i].spawnCar();
      }
    }

    // Reset graphEdges car count
    for (int i = 0; i < graphEdges.size(); i++) {
      graphEdges[i].numCarsPresent = 0;
    }

    // Run A* for newly spawned cars and count number of Cars on each Edge
    for (auto it = graphCars.begin(); it != graphCars.end(); it++) {
      int id = it->first;
      Car& car = it->second;
      if (car.roadIndex == -1) {
        car.route = astar(*car.start, *car.end);
        car.currentRoad = car.getNextRoad();
        car.currentRoadDistance = 0.0;
      }
      graphEdges[car.currentRoad->id].numCarsPresent++;
    }

    // Update car speeds
    for (int i = 0; i < graphEdges.size(); i++) {
      graphEdges[i].updateActualSpeed();
    }

    vector<int> toErase;

    // Update car positions and check if Cars have reached destination
    for (auto it = graphCars.begin(); it != graphCars.end(); it++) {
      int id = it->first;
      Car& car = it->second;
      ld timeLeft = TIME_STEP;
      while (timeLeft > 0.0) {
        ld roadTime = (car.currentRoad->length - car.currentRoadDistance) / car.currentRoad->actualSpeed;
        if (timeLeft >= roadTime) {
          car.distanceTraveled += car.currentRoad->length - car.currentRoadDistance;
          timeLeft -= roadTime;
          if (car.currentRoad->end == car.end) { // Car has reached destination
            ld tripTime = car.getTimeElapsed() + (TIME_STEP - timeLeft);
            TOTAL_TRIP_TIME += tripTime;
            TOTAL_TRIP_COUNT++;
            CURRENT_CAR_COUNT--;
            toErase.pb(id);
            car.finished = true;
            break;
          } else {
```

```
          car.currentRoad = car.getNextRoad();
          car.currentRoadDistance = 0.0;
        }
      } else {
        // cout << "crd " << car.currentRoad->actualSpeed * timeLeft << endl;
        car.distanceTraveled += car.currentRoad->actualSpeed * timeLeft;
        car.currentRoadDistance += car.currentRoad->actualSpeed * timeLeft;
        timeLeft = 0.0;
      }
    }
    if (car.finished) {
      continue;
    }
    // cout << convertToString(car) << endl;
  }

  // Remove Cars that have arrived at their destination
  for (int key: toErase) {
    graphCars.erase(key);
  }

  // Increment time
  CURRENT_TIME += TIME_STEP;
  // cout << endl;
}
```