A Simulation-Based Assessment of Traffic Circle Control

MCM Team #5702

February 9, 2009

Technical Summary

Evaluating the performance of traffic control systems for traffic circles and determining the most effective traffic control system to use for a specific circle is currently an active area of research. The difficulty of this problem lies largely in its crucial dependence on the local interactions between individual drivers. In particular, traffic circles are relatively small (when compared to highways, for instance) and are therefore susceptible to blockages caused by the actions of individual cars such as lane changes, entrances, and exits. A complete model, then, must account for the effects of individual car behavior. Existing models of large traffic circle behavior, however, do not track performance at the level of individual cars.

We propose here a novel simulator-based approach to evaluating and selecting traffic control systems for traffic circles. We create a multi-agent, discrete time simulation of traffic circle behavior under different traffic control systems. The behavior of individual cars in our simulator is determined *autonomously* and *locally*, allowing us to capture the effects of their local interactions. In addition, by modeling each car separately, we are able to track the time spent in the traffic circle for each individual car, giving us a more specific measure of traffic circle performance than the more commonly used aggregate rate of car passage.

Measuring the performance of several different traffic control strategies using these two metrics, we found that the **rate of incoming traffic** and the **number of lanes** in the traffic circle were the major driving factors behind the optimal choice of traffic control system. Based on the simulated performance of traffic circles with varying values of these parameters, we have two different recommendations for traffic control systems based upon the rate of incoming traffic.

When the rate of incoming traffic is low, we recommend that **entering cars yield to cars already in the circle**. When the rate of incoming traffic increases beyond a certain threshold, which should be determined empirically, we recommend the use of **traffic lights**

that control entering traffic and the outermost lane of the traffic circle. These lights should be synchronized so that the time between successive lights turning green is the average time needed for a car to travel between them.

In the case with a low rate of incoming traffic, the circle is relatively clear of cars. Therefore, cars entering the circle are able to merge in without blocking the road and slowing down the flow of traffic. By making entering cars yield to cars in the circle, we are able to maximize the total rate of cars passing through while maintaining the average speed of cars.

When the rate of incoming traffic increases to saturated the circle with cars, allowing cars to merge freely into the circle creates gridlock within the circle. In particular, the presence of so many cars attempting to enter and exit impedes the flow of the others. While the throughput of this system is still quite high, our simulation predicts that each individual car will spend an extremely long time traversing the traffic circle.

Instead, we recommend that traffic lights be used to attenuate the incoming flow of cars. In this case, while cars must wait slightly longer to enter the circle, the number of cars in the circle is limited, allowing cars within the circle to travel at a reasonable speed. Our simulator predicts that this will allow fewer cars to travel through the circle at a much higher speed. By viewing the performance of the control system at the level of the individual cars, our simulator is able to distinguish between the performance of these two systems in this case and select the correct system to use.

Given a traffic circle, then, we recommend using our conclusions in the following way. First, measure the rate of incoming traffic incident at the circle during various times of day and examine the occupancy level of the circle under the outer yield control system. For time periods with high occupancies and rates of incoming traffic, we recommend implementing synchronized traffic lights. For the other time periods, we recommend requiring the entering roads to yield to the cars in the circle. Under this system, the total throughput of the traffic circle is maximized, while still maintaining an acceptable level of individual performance.

Contents

1 Introduction

The traffic circle is a type of circular intersection featuring traffic circulating from multiple streets circulating around a central island, usually in one direction. In particular, traffic flows in a circle around a central island in one direction only, and the roads running into and out of the circle intersect it at different locations. An example of a traffic circle is shown below [1]:

Figure 1: An aerial view of Dupont Circle in Washington, DC.

Other examples of large traffic circles include Columbus Circle in New York City, while small, one-lane traffic circles often exist in residential neighborhoods. Traffic circles are often notorious for the frequent traffic jams that occur due to their unconventional design, and many different methods exist to control traffic within a traffic circle. Different circumstances for the use of traffic circles seem to require different types of traffic controls, and it therefore seems worthwhile to investigate the impact of the different approaches.

1.1 Terms and Notation

In this paper, we will consider a traffic circle to be a one-way circular road with two-way roads meeting the circle at T-junctions. In particular, we will not consider circles with separate entry and exit ramps. We will assume that each road carries cars into the circle at a fixed rate and that cars have an equal probability of leaving the circle through any of the other roads. In measuring performance of the traffic circle, we will measure two statistics—the average rate at which cars arrive at their desired exit location per time step, which we will

denote as the *average throughput*, and the average number of timesteps between when a car arrives at the back of the queue to enter the circle and when it exits the circle, which we will simply denote as the *average total time*.

1.2 Problem Background

Modern traffic circles have recently been recognized as safer alternatives to traditional intersections. Research by Zein et. al. [18] and Flannery et. al. [8] using statistical methods has demonstrated the added safety traffic circles bring to both urban and rural environments. Attempts to understand the specific safety and efficiency benefits of traffic circles have taken four primary approaches: critical gap estimation, regression studies, continuous models, and discrete models.

Critical gap models build off of how drivers empirically gauge gaps in traffic before merging or turning into a traffic stream. However, according to Brilon et. al., attempts that began in the 1980s to model roundabout capacity based off gap acceptance theory were not exceptionally promising [5]; in particular, Briton et. al. claimed critical gap estimation lacked valid procedures as well as general clarity [6]. More recent research applying gap-acceptance models to understanding traffic circles has included Polus et. al. [16] and the modeling of unconventional traffic circles by Bartin et. al. [2].

Nevertheless, regression studies on empirical data made much progress, beginning in 1980 with [12], where Kimber studied roundabouts in England and discovered a linear relation relating entry capacity to circulating flow and constants depending on entry width, lane width, the angle of entry, and the traffic circle size. Further regression studies have built extensively off of Kimber's work, such as in [15], which determined the importance of traffic circle diameter in small-to-medium circles.

In addition to regression studies, research in modeling traffic flow has also progressed from a continuous approach to modeling vehicular traffic, e.g. Helbing with improved fluid-dynamic models [10], Bellomo et. al. with fluid dynamic models that followed the Boltzmann equations [3], Daganzo with improvements on fluid-based models [7], and Klar et. al. with homogeneous and inhomogeneous traffic flow [13]. However, the four aforementioned papers model traffic flow in standard traffic environments apart from traffic circles.

Additional research has also included discretized approaches such as cellular automata models [9, 13] and discrete stochastic models [17]. Discrete models are more suitable for research involving small environments like traffic circles, where studying individual car-to-car interactions takes an interest over generalizing traffic flow as a whole.

Furthermore, discretized approaches have attempted to model multi-lane traffic flows [14]. However, despite the results of the research described above, to our knowledge there has been no research done on discrete models of multi-lane traffic circles of varying sizes.

1.3 Our Results

In this paper, we approach the problem of traffic circle control by first creating a simulator that models the traffic flow around a given circle. The simulator treats individual cars as autonomous units, allowing us to capture local interactions such as lane changes and traffic

blockages due to cars entering and exiting. We validated this simulator against both a new stylized model of the situation and existing models of traffic circle flow.

Using this simulator, we implemented and tested different control systems on different types of traffic circles. Based on the simulated results, we were able to isolate the **rate of incoming traffic** and the **number of lanes** in the traffic circle as the major driving factors behind the optimal choice of traffic control system. We were thereby able to recommend two different systems for different circumstances. When the rate of incoming traffic is low, we recommend that **entering cars yield to cars already in the circle**. When the rate of incoming traffic increases beyond a certain threshold, we recommend the use of **traffic lights that control entering traffic and the outermost lane of the traffic circle**. These lights should be synchronized so that the time between successive lights turning green is the average time needed for a car to travel between them.

The rest of this paper is organized as follows. In Section 2, we divide the problem into two portions and define our objectives for each. In Section 3, we introduce a simulator for the performance of a traffic circle and validate its performance against a mathematical analysis and models from other sources. In Section 4, we use our simulator to analyze the performance of several types of traffic circles to produce recommendations for which control systems should be used for each type. In Section 5, we provide an overview of the advantages and disadvantages of our approach and give some directions for future work.

2 Problem Setup

In this paper, we approach the problem in two portions. First, we create a discrete simulator to model the behavior of cars in a traffic circle based on local interactions at the level of each car. We then use this simulator to evaluate the performance of the various techniques of traffic control and to determine which method should be used in each specific circumstance. Before proceeding, however, we specify our objectives in each part a bit more closely.

2.1 Simulator

Our goal in this part of the paper is to create a simulator that, given a set of conditions and traffic rules, can produce an accurate prediction of the behavior that will result from following these rules. To achieve this goal, we would like our simulator to fulfill the following requirements:

1. The simulator takes into account the local interactions between cars.

   Because traffic circles are often occupied by many cars that enter, exit, and change lanes quite frequently relative to the size of the traffic circle, such interactions between cars make a major contribution to the speed and efficacy of any traffic circle. Any effective simulator must therefore take these effects into account.

2. The simulator can support variation in the number of cars, size of the circle, and number of lanes.

   In order to make predictions for the many different types of traffic circle extant, it is necessary to create a simulator that will work for all of them.

3. The simulator can track properties of both the entire traffic circle and the individual cars passing through it.

   Because individual interactions are so important in determining the behavior of traffic circles, we must evaluate the results of a specific control system not only at a global level but also for each individual.

For each of these cases, while the real behavior of cars in a traffic circle may vary widely, we wish to capture the most essential aspects of a control system for a particular traffic circle. Therefore, we restrict our simulator to dealing with idealized behavior of cars. In particular, we will assume that all cars in our simulation follow the traffic regulations that we have put into place. In addition, we will assume that no accidents or crashes happen.

2.2 Control System Evaluation

The ultimate objective of this paper is to apply the previously described simulator to produce concrete recommendations for which method of traffic control should be chosen for a specific traffic circle. We do so based upon the following statistics:

1. The average throughput (the average rate at which cars pass through the traffic circle).
2. The average number of cars in the traffic circle.
3. The average total time it takes for each car to traverse the traffic circle, including time spent waiting to enter.
4. The average time each car spends driving through the traffic circle.

Note that statistics 1 and 2 measure global properties of the traffic circle, while statistics 3 and 4 are properties of each individual car. To evaluate the efficacy of a control system, then, we must consider both the global performance of the system and the differences in the performance of the system for each individual. In particular, our goals are to:

1. Maximize the average throughput of the traffic circle.
2. Minimize the total time spent traversing the traffic circle (for individual cars).

Here, Goal 1 gives a measure of the overall performance of the system, which we of course wish to maximize. However, while doing so, it is desirable for the system to avoid bad performance for individual cars, which is given by Goal 2. For the rest of this paper, then, we will evaluate the performance of a traffic circle by **the rate of cars passing through the circle** (*average throughput*) and **the total time required to traverse the circle** (*average total time*). Our goal will be to choose the traffic control method that performs best with respect to both of these metrics.

3 Simulator

In this section, we describe a discrete traffic simulator used to predict the behavior of a traffic circle under a given set of traffic rules.

3.1 Assumptions and Setup

To model the behavior of traffic under certain conditions, there are essentially two possibilities:

1. Make a (usually continuous) abstraction away from the discrete interactions of cars and deal with a more stylized model of the entire system.

2. Model the behavior and movement of each car separately.

Many studies of traffic flow model traffic behavior as continuous and fluid-like, as in the first possibility. Such models are suitable under a macroscopic view of traffic, for instance in the study of traffic on long roads or highways. However, for intersections and traffic circles where specific car-to-car interactions occur much more frequently, a continuous fluid model seems inadequate.

In this paper, we follow the first second possibility to model traffic flow in a traffic circle using a multi-agent discrete time simulation. Our simulation is based around the following two key principles:

1. It is microscopic.

2. Behavior and information is local.

We do not use an abstract view of traffic as a flow, but instead let each car in the traffic circle be its own individual agent. This allows us to account for the effects of car-to-car interaction, particularly in congested situations. From this interaction on the microscopic level, we then examine the macroscopic consequences of the simulation, instead of beginning with an arbitrary conception of what the macroscopic behavior should be.

Each car is its own independent agent, trying to enter the traffic circle and exit at the desired exit as quickly as possible; no collaboration between cars or higher level organizational principles exists. Also, only local information, namely the cars in the immediate neighborhood, is available to each individual car.

3.2 The Simulator

Our approach will be to develop a simulation that realistically captures driver behavior in traffic circles and examine the consequences of this behavior on traffic flow. Specifically, the simulation operates using the following model:

- Time is modeled in discrete timesteps.

- The traffic circle is a rectangular grid. The width of the grid is the number of lanes in the traffic circle, and the height of the grid represents the length of the traffic circle. The upper edge of the grid wraps around to the lower edge of the grid (so that the grid is actually a circle). At any time step, each square of the grid can either be empty or hold one car.

- Certain squares in the right-most lane are designated as entry squares. A queue of cars waits at each entry square to enter the traffic circle. (These cars are not located on the grid itself.) The queues start off empty, and for each entry square, there is a fixed probability of a car being added to the queue at each time step.
- Certain squares in the right-most lane are designated as exit squares. Cars can exit the traffic circle at these exit squares. When a car is added to the queue for an entry square (and thus to the system), an exit square is chosen at random as the location at which the car will wish to exit the traffic circle.
- Each car has a speed indicated by how often it gets the chance to move. E.g., faster cars may get the chance to move at every time step while slower cars may only move at every other or every third time step. This difference in speed can simulate the differing levels of impatience or aggression among drivers.

In each timestep, the simulation proceeds as follows:

1. Determine the subset of all the cars in the system that will move during this timestep. Randomly assign the order in which these cars will move.
2. Allow each such car to move. Cars move under the following rules:
   A car that is already in the traffic circle at position (i, j) (i.e. lane i, vertical position j) will, in the following order of preference,
   (a) Exit if (i, j) is the exit square at which the car wishes to exit.
   (b) Move forward to $(i, j + 1)$ if $(i, j + 1)$ is unoccupied.
   (c) Move forward and right to $(i+1, j+1)$ if there is a lane to the right and locations $(i + 1, j)$ and $(i + 1, j + 1)$ are both unoccupied.
   (d) Move forward and left to $(i - 1, j + 1)$ if there is a lane to the left and locations $(i - 1, j)$ and $(i - 1, j + 1)$ are both unoccupied.
   (e) Stay where it is.

   An exception occurs for cars that are about to exit—if the vertical distance between the car's current location and its desired exit is less than 4 times the horizontal distance, then the second and third items above are switched. (This is to ensure that under non-congested situations, cars will be able to exit at their desired exits.)
   A car that is the first car in the queue at an entrance location will, in order of preference,
   (a) Move to the entry square if that square is unoccupied.
   (b) Stay where it is.

   All later cars in the queue cannot move for this turn.
   In addition to the above rules, the specific traffic control systems impose the following additional rules:

(a) Outer-yield: In this system, a car at the front of an entrance queue and waiting to enter can only enter when both the entry square and the square directly behind the entry square are empty. That is, if there is a car directly behind the entry square, the entering car must yield to that car.

(b) Inner-yield: In this system, if a car in the circle wishes to move onto an entry square (in the rightmost lane) but the queue waiting at that entrance is non-empty, then the car cannot move to that square. If the car has no other possible moves, then it does not move for that turn. This reflects the situation in which cars in the circle need to yield to entering cars.

(c) Traffic lights: In this system, a traffic light controls each entry square. At any time step, the light is either green for cars in the circle and red for the waiting queue, or vice versa. If it is green for cars in the circle, then the first car in the waiting queue cannot enter the circle. If it is green for the waiting queue, then no car in the circle can move onto the entry square. Note that in a multi-lane circle, this traffic light only controls traffic in the rightmost lane. This behavior is inspired by the design of metering lights at highway ramps.

For the traffic light control system, we consider two different methods of synchronizing the traffic lights around the circle. In the first method, all lights to turn green and red simultaneously. In the second method, the difference in time between each traffic light turning green and the next light turning green (and also the difference in time between this light turning red and the next light turning red) is directly proportional to the distance between these two lights. The proportionality constant is chosen so that a car that was waiting at a traffic light and begins to move when that light turns green will reach the next light just as it turns green.

3. For each entry queue, add a car to the end of that queue with the fixed probability for that entry location.

4. Flip the traffic lights if it is the correct timestep to do so.

3.3 Validation Against Existing Empirical Models

The two criteria on which we evaluate the various traffic control systems, average throughput and average total time, are not unrelated. In fact, our simulations indicate that increasing one comes at the cost of increasing the other. Therefore, we would like to validate the accuracy of our simulation by examining the relationship produced by our simulation. In simulations on a fixed traffic circle under the outer yield system, the plot of reserve throughput (maximum throughput minus average throughput) against average total time over variations in incoming car density is shown below:

This inverse relationship is clear intuitively, as a higher amount of throughput indicates a greater volume of traffic on the road and hence both slower driving speeds and longer wait times to enter the circle. This result also matches the relationship between average total time and reserve capacity given by the Kimber-Hollis delay equation in [4]. This rough correlation indicates that the results of our simulator are relatively reasonable.

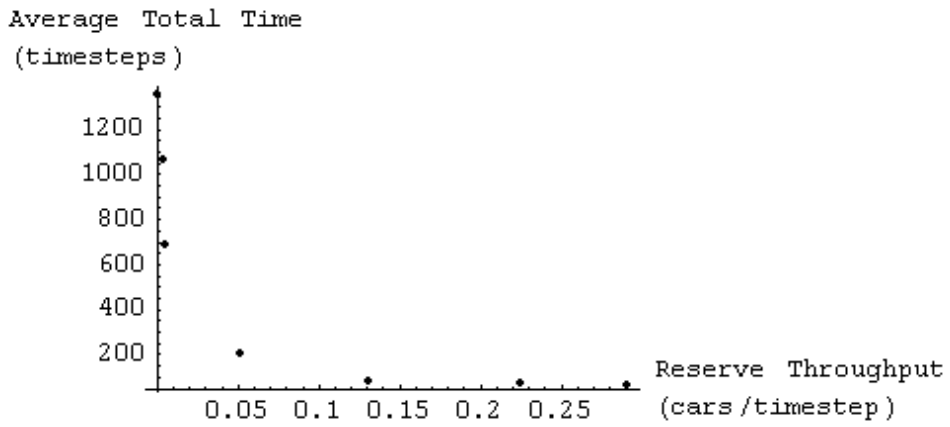


Figure 2: Average Total Time vs. Reserve Throughput

3.4 Validation Against a Simple Model

To provide further verification of the accuracy of our simulator, we compare the large scale features of its output to a mathematical model for a simple case. In particular, we will consider a single-lane traffic circle in which cars entering the circle need to yield to cars already in the circle. For simplicity, we assume that all cars move at the same speed of one square per time step. We assume that roads leading to the traffic circle are all two-way roads, so that each entry point of the traffic circle is also an exit point. The model is given as follows:

Suppose that there are n entry/exit roads to the traffic circle, and that all cars have an equal probability of wanting to leave through each of the n roads. For $i = 1$ to n, let r_i be the probability that a new car appears at road i at any time step. Let x_i be the volume density of traffic in the segment of the circle between roads i and $i+1$. The expected change in the number of cars between roads i and $i+1$ is given by a sum of four terms:

- The probability that a car will leave this segment through exit $i+1$ is $x_i \cdot \frac{1}{n}$, as x_i is the probability that there is a car in the exiting square and $\frac{1}{n}$ is the probability that this car wishes to exit.

- The probability that a car will move from this segment to the next segment is $x_i \cdot \frac{n-1}{n} \cdot (1 - x_{i+1})$, as $\frac{n-1}{n}$ is the probability that the car in the exiting square will not exit and $1 - x_{i+1}$ is the probability the the square after the exiting square, which is the first square of the next segment, is unoccupied.

- The probability that a car will move from the previous segment to this segment, similarly, is $x_{i-1} \cdot \frac{n-1}{n} \cdot (1 - x_i)$.

- The probability that a car will enter through entrance i is the probability p of there being a sufficiently large space at entrance i for a car to enter, times the probability that there is a car waiting to enter at that entrance. This latter probability can be computed as $r_i + r_i(1-r_i)(1-p) + r_i(1-r_i)^2(1-p)^2 + \cdots$ since there is an r_i probability of a car

arriving at entrance i this time step, an $r_i(1-r_i)(1-p)$ probability of a car arriving at entrance i last time step (but not this time step) and remaining until this time step, etc. This sum is $\frac{r_i}{r_i+p-r_ip}$. We note that in our simulation, $p=(1-x_{i-1})(1-x_i)$ as a car can enter into the circle if the entrance point and the previous square are unoccupied.

So the expected change in the number of cars in this segment in one time step is

$$\Delta c_i = -x_i \cdot \tfrac{1}{n} - x_i \cdot \tfrac{n-1}{n} \cdot (1-x_{i+1}) + x_{i-1} \cdot \frac{n-1}{n} \cdot (1-x_i) + \tfrac{r_i(1-x_{i-1})(1-x_i)}{r_i+(1-x_{i-1})(1-x_i)+r_i(1-x_{i-1})(1-x_i)}.$$

In equilibrium, this change should be 0 for all segments, giving a system of equations in the x_i. If we consider the case where the roads have equal incoming traffic, i.e. r_i is the same for all i, then by symmetry the x_i are the same for all i, and we may solve the equation

$$\Delta c = -x \cdot \tfrac{1}{n} - x \cdot \tfrac{n-1}{n} \cdot (1-x) + x \cdot \tfrac{n-1}{n} \cdot (1-x) + \tfrac{r(1-x)^2}{r+(1-x)^2+r(1-x)^2} = 0$$

numerically for x in terms of r. Here, x is the traffic volume density for the circle as a function of r, the rate at which cars enter the circle through each road. The result of numerically solving for x is shown in the figure below together with the corresponding plot generated by our simulator:

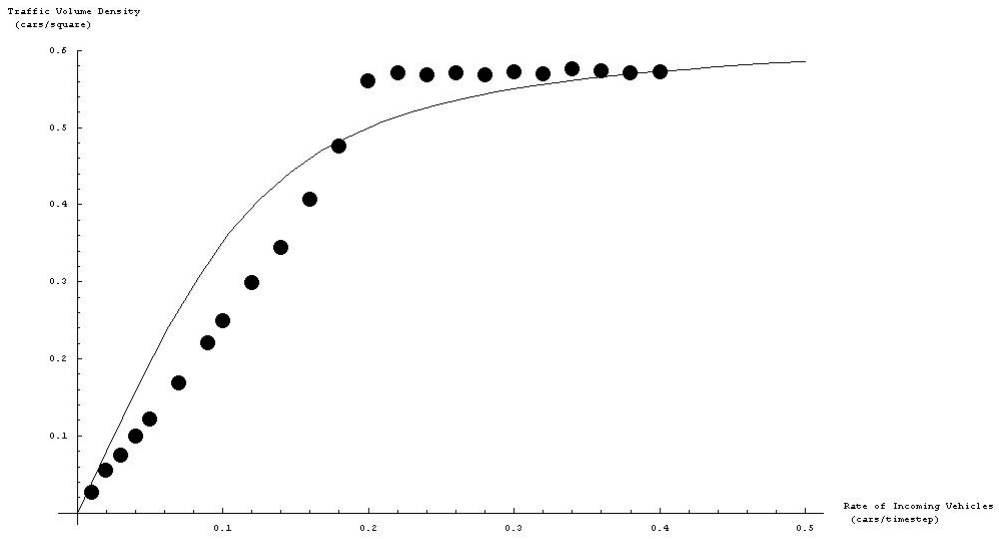


Figure 3: Traffic Volume Density vs. Rate of Incoming Vehicles

Here, the black data points were generated by our simulator, and the curve was produced by the rudimentary model. The volume density of both seems to grow in a somewhat linear fashion for low rates of incoming vehicles. When the number of incoming vehicles increases, the traffic circle appears to become saturated at a fixed density. As can be seen, the simulator and our mathematical model agree on these large-scale features. While there is some disagreement around the critical rate of incoming vehicles, this might be explained by the fact that our mathematical model essentially considered the cars in each segment as equivalent, which ignores the small scale interactions that occur near gridlock.

4 Predictions and Analysis

In this section we apply the simulator to analyze different types of traffic circles.

4.1 Criteria

We will characterize traffic circles by the following variables which are fixed for each traffic circle and which we therefore take as given when deciding between traffic control systems.

1. **Rate of Incoming Vehicles:** This is a result of the amount of traffic present on the roads entering the traffic circle and will influence the total number of vehicles trying to enter the circle and hence the traffic in the circle.

2. **Length:** This affects the number of cars that can be contained in the circle at a single time, which has many implications for how the entry mechanism of the circle should be determined.

3. **Number of Lanes:** This affects both the number of cars that can be in the circle at a time and their maneuverability around each other. Because cars can more easily pass one another with more lanes, increasing the number of lanes may reduce the effects of traffic blockages.

4. **Number of Incoming Roads:** This affects the rate at which cars need to enter and exit the traffic circle, which may influence the magnitude of traffic blockages.

When considering different traffic control systems, we wish to consider systems that are relatively close to conventional traffic control systems, as it would be impractical and hazardous to introduce radically different systems that may be unfamiliar for drivers who do not encounter traffic circles very frequently. Therefore, we will evaluate the performance of the following traffic control systems when we vary the parameters for our traffic circle:

1. **Outer Yield:** Cars attempting to enter the circle yield to cars already in the circle at all times.

2. **Inner Yield:** Cars already in the circle yield to cars attempting to enter the circle.

3. **Simultaneous Lights:** The intersections between the circle and other roads are controlled by traffic lights which all turn green and red at the same time. However, the traffic lights only apply to the outermost lane of the traffic circle.

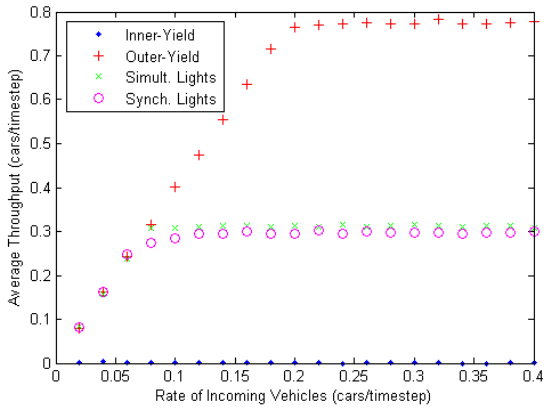
4. **Synchronized Lights:** The intersections between the circle and other roads are controlled by traffic lights where the time interval between a light turning green and the next light turning green is proportional to the distance between the two lights. Here, the traffic lights only apply to the outermost lane of the traffic circle.

With the exception of the traffic lights, these control systems are all similar to existing traffic control systems. However, there is a crucial difference between our traffic light system and standard traffic lights, as only stopping the outer lane of the traffic circle allows traffic in the inner lane to proceed undisturbed, improving the throughput of cars in the circle. This approach is a hybrid of normal traffic lights and metering lights for congested highways.

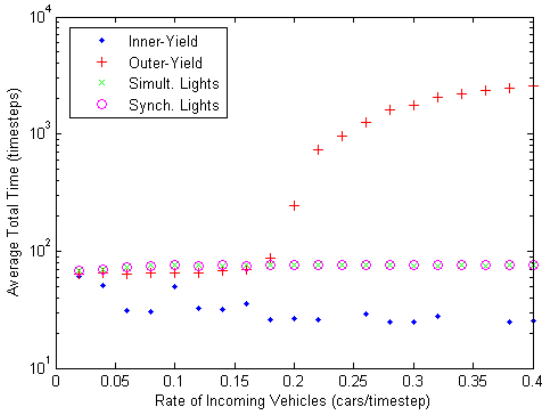
4.2 Analysis

To analyze the effects of the different traffic control systems on the different types of traffic circles, we created different traffic circles by varying our parameters. We then ran each traffic control strategy on these different circles and created plots of the average throughput and average total time per car for each strategy. In particular, our goal through these simulations was to determine which of the parameters had the greatest effect on the simulated performance of the different traffic control systems.

Changing the different parameters, we produced the following plots measuring the effectiveness of the various traffic control systems:

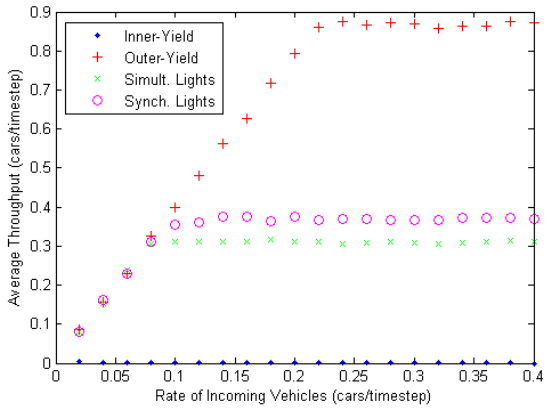


Average Throughput vs. Rate of Incoming Vehicles

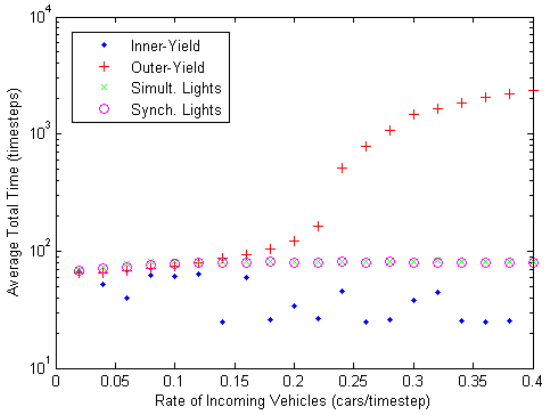


Average Total Time vs. Rate of Incoming Vehicles

Figure 4: Performance for 1 lane, length 100, 4 roads, rate variable.

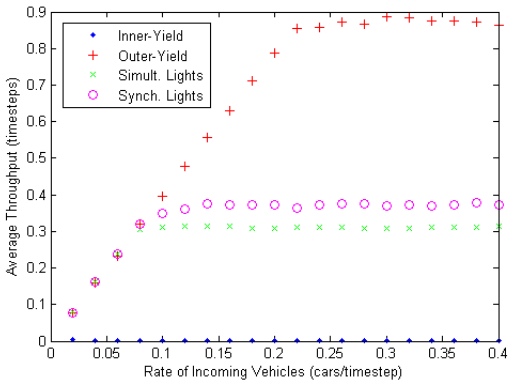


Average Throughput vs. Rate of Incoming Vehicles

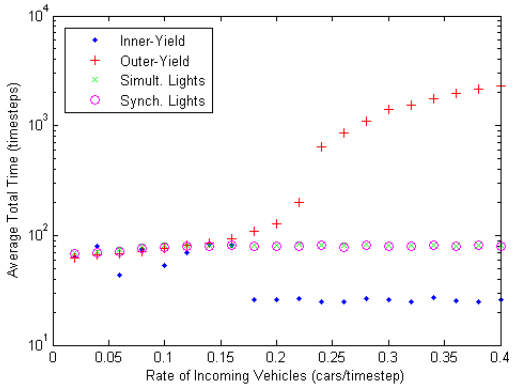


Average Total Time vs. Rate of Incoming Vehicles

Figure 5: Performance for 3 lanes, length 100, 4 roads, rate variable.

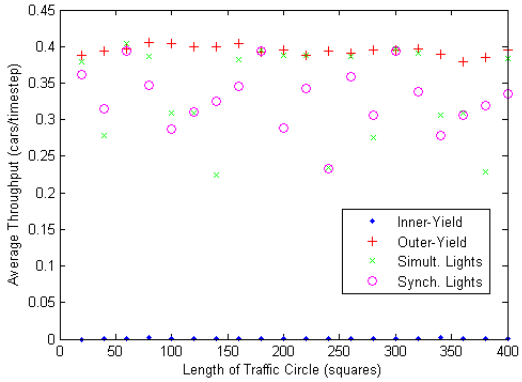


Average Throughput vs. Rate of Incoming Vehicles

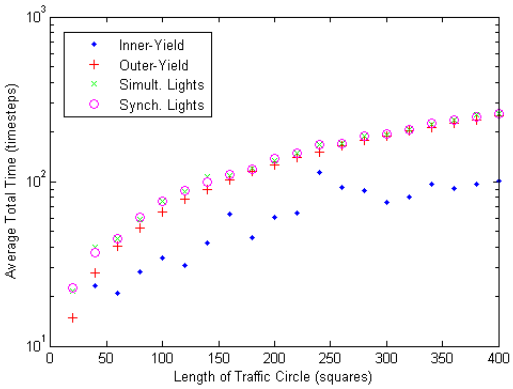


Average Total Time vs. Rate of Incoming Vehicles

Figure 6: Performance for 5 lanes, length 100, 4 roads, rate variable.

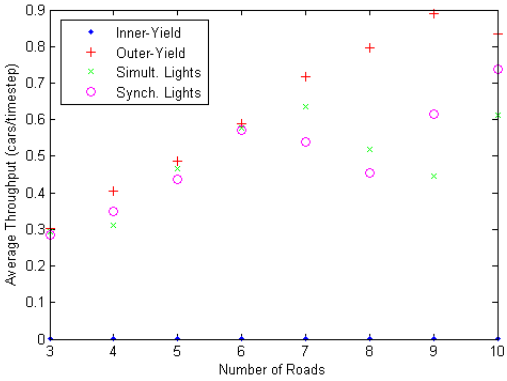


Average Throughput vs. Length

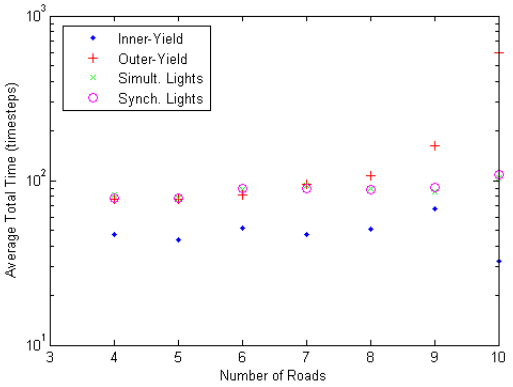


Average Total Time vs. Length

Figure 7: Performance for 3 lanes, rate 0.1, 4 roads, length variable.



Average Throughput vs. Number of Roads



Average Total Time vs. Number of Roads

Figure 8: Performance for 3 lanes, rate 0.1, length 100, roads variable.

From these plots, we can make the following observations:

- In almost all the plots, the inner yield system has almost no throughput, as the cars in the road become gridlocked because they too often yield to incoming cars and therefore cannot exit. Note that the low value of the average total time for this control system results from the fact that the only cars that are able to exit do so before the road becomes entirely gridlocked. As a result, we can reject the inner yield system as a possible control system in any circumstance.

- As the rate is varied in the first three sets of plots, the throughput of the outer yield system and the traffic light systems correspond for small values of the rate. However, for each system, the throughput reaches a plateau after a certain value of the rate. At this point, it appears that the traffic circle has been saturated, meaning that it can no longer accept more cars from the incoming roads.

- The throughput value at which saturation occurs is much higher for the outer yield system, meaning that more cars can pass through the circle in a given time period under this system. However, the amount of time required for each individual car to pass through the traffic circle under the outer yield system is extremely high, almost an order of magnitude higher than needed under either traffic light system.

- When there are either 3 or 5 lanes, the synchronized traffic light system allows slightly greater throughput than the simultaneous traffic light system. This might be explained by the fact that, with more lanes, cars can move in a more uniform manner, allowing them to use the synchronized lights and move through the circle more quickly.

- The number of lanes and the number of roads do not seem to have a significant effect on either the throughput or total time of the outer yield system or either of the traffic light systems. However, it appears that the traffic lights may perform worse for some values of the distance between roads, perhaps due to synchronization issues.

In general, it appears that the outer yield and traffic light methods have an advantage over the inner yield method, and that the correct choice of control system is largely determined by the rate of incoming vehicles on each of the roads which meet at the circle.

4.3 Recommendations

Based on the observations we made in the previous section, we would like to give concrete recommendations for what traffic control system to use for a specific traffic circle. As we noted, the number of lanes and the number of roads incident at the traffic circle do not seem to significantly affect the average throughput of the circle or the average time required for each car to traverse the circle. We may therefore restrict our attention to the rate of incoming cars and the number of lanes in the circle.

Of these two variables, the rate of incoming cars accounts for a large part of the variation in performance, as can be seen in Figures 4, 5, and 6. For values of this rate between 0 and 0.1 cars per timestep, the performance of the outer yield and traffic light systems is identical, as in this range traffic is light and there is very little interaction between cars.

As the rate increases to between 0.1 and 0.2 cars per timestep, the traffic circle reaches its maximum throughput under the traffic light system, while the average total time stays fixed. However, under the outer yield system the throughput of the system continues to increase at the cost of a rather dramatic increase in the average total time. In this range, choosing between the outer yields system and the traffic light systems involves a tradeoff between the throughput, or the *quantity* of cars passing through, and the total time, or the *speed* at which cars can pass through.

Finally, as the the rate increases above 0.2 cars per timestep, the traffic circle becomes saturated with cars, meaning that the average total time for the outer yield system increases dramatically. In this case, this circle experiences gridlock with the outer yield system, meaning that cars move extremely slowly and must wait a very long time to pass through. Under the traffic light systems, however, a smaller number of cars can pass through, but the average total time required for them to pass remains similar to that required for a much lower rate. As the inner yield system requires an extremely large total time in this range, the traffic light systems are clearly superior for a rate of above 0.2 cars per timestep.

Observe that in each of these cases, the synchronized traffic lights allowed for higher throughput than the simultaneous traffic lights, meaning that they should always be preferred.

We can now make the following recommendations for the control of traffic circles:

- **For a low rate of entering cars, no traffic lights should be used in the traffic circle.** Instead, cars already in the circle should be given the right of way, and cars entering the circle should yield.

- **As the rate of entering cars increases, the use of synchronized traffic lights for the outer-most lane of the traffic circle only should be considered** in order to ensure a reasonable time of traversal for most cars.

- **For large rates of entering cars, as may occur during rush hour, synchronized traffic lights should be used** in order to ensure that the traffic circle does not become congested. By preserving a reasonable flow of cars within the circle, synchronized traffic lights allow a slightly smaller number of cars to pass through the circle much more quickly, which is definitely preferable to deadlock for all cars.

In the low and high limits of the rate, our recommendation seems to correspond to similar practices elsewhere. Few traffic lights are deemed necessary when there are a small number of cars present in the low limit. In the high limit, it is a common practice in California to use metering lights to limit the number of vehicles that can merge onto a highway during peak hours in order to ensure that the cars already on the road can move at a reasonable speed. Our recommendations then, seem to be a mix of these two ideas applied to traffic circles.

5 Conclusions

5.1 Strengths

Our simulator takes into account the behavior and outcomes of individual cars traveling through a traffic circle. By doing so, we are able to detect interactions at a microscopic level and to track the performance of a traffic control system for each individual rather than only in aggregate. This allows our model to evaluate the effects of cars changing lanes and entering and exiting from specific lanes. The simulator was validated against both an existing empirical model and the results of a simple model for the steady-state limit.

Using this simulator, we were able to simulate the performance of a widely varied spectrum of traffic control systems on a range of different traffic circles. Our results allowed us to isolate the rate of incoming cars and the number of lanes in the traffic circle as the two parameters key to determining the correct control system.

By analyzing the performance with respect to these parameters, we recommended the use of either an outer yield system or synchronized traffic lights to control the traffic circle, depending on the rate at which cars enter the circle. These recommendations based on evidence from our simulator also correspond to traffic control systems used for similar practices, suggesting that our changes to them may also be effective when applied to traffic circles.

5.2 Weaknesses

While our simulator attempted to model the behavior of drivers fairly accurately, it was of course impossible to completely capture the dynamics of lane changing and breaking. Further, while using a discrete time, discrete space model for the simulator allowed us to capture the local, multi-agent nature of individual drivers, it forced us to make some simplifications with regards to the continuity of car movement and with regards to simultaneous actions.

In addition, our simulation did not take into account the fact that, in an actual traffic circle, the inner lanes have slightly shorter length than the outer lanes, as we are assuming that the circle is large enough so that this does not occur.

We only considered traffic lights with simultaneous or synchronized light changes, and it was infeasible computationally for us to consider a wider variety of light switching approaches. While using synchronized lights did allow us to improve upon the throughput of regular traffic lights, further improvements might be available by examining other possibilities.

5.3 Alternative Approaches and Future Work

There were several extensions to our simulator that we could not pursue due to the time constraint. We considered evaluating the safety of a particular traffic control system by counting the number of conflicting desired movements at the local level. We would then be able to compare systems by safety as well as pure performance, and in particular we might be able to evaluate the claim that certain types of traffic circles are safer than intersections [8].

A Source Code for Simulator

The source code for the simulator has been appended to this paper.

References

[1] Aerial photograph of Dupont Circle in Washington, D.C., USA. Taken by the United States Geological Survey. Available at http://en.wikipedia.org/wiki/index.html?curid=1017545.

[2] Bartin, Bekir, Kaan Ozbay, Ozlem Yanmaz-Tuzel, and George List. "Modeling and Simulation of Unconventional Traffic Circles." Transportation Research Record: Journal of the Transportation Research Board, V. 1965 (2006).

[3] Bellomo, N., M. Delitala, V. Coscia, and F. Brezzi. "On the Mathematical Theory of Vehicular Traffic Flow I: Fluid Dynamic and Kinematic Modelling." Mathematical Models & Methods in Applied Sciences, V. 12 (2002).

[4] Brilon, Werner and Mark Vandehey. "Roundabouts—The State of the Art in Germany." Institute of Transportation Engineers (ITE) Journal, V. 68 (1998).

[5] Brilon, Werner, Ning Wu, and Lothar Bondzio. "Unsignalized Intersections in Germany—a State of the Art." Published in the Proceedings of the Third International Symposium on Intersections without Traffic Signals, Portland Oregon (1997). Available at http://www.ruhr-uni-bochum.de/verkehrswesen/vk/deutsch/Mitarbeiter/Brilon/Briwubo_2004_09_28.pdf.

[6] Brilon, Werner, Ralph Koenig, and Rod J. Troutbeck. "Useful Estimation Procedures for Critical Gaps." Transportation Research Part A (1999). Available at http://www.sciencedirect.com/science/article/B6VG7-3VF9D7R-2/2/2b325096a3b448fd0c0a09952c091ff4.

[7] Daganzo, Carlos F. "Requiem for Second-Order Fluid Approximations of Traffic Flow." Transportation Research Part B: Methodological, V. 29 (1995). Available at http://www.sciencedirect.com/science/article/B6V99-3YKKJ1D-F/2/f9d735df36de4048d1e62a0a20e844b0.

[8] Flannery, Aimee and Tapan K. Datta. "Modern Roundabouts and Traffic Crash Experience in United States." Transportation Research Record: Journal of the Transportation Research Board, V. 1553 (1996).

[9] Fouladvand, M. Ebrahim, Zeinab Sadjadi and M. Reza Shaebani. "Characteristics of Vehicular Traffic Flow at a Roundabout." Phys. Rev. E 70, 046132 (2004).

[10] Helbing, Dirk. "Improved fluid-dynamic model for vehicular traffic." Phys. Rev. E, V. 51 (1995).

[11] Hossain, M. "Capacity Estimatin of Traffic Circles under Mixed Traffic Conditions using Micro-Simulation Technique." Transportation Research Part A: Policy and Practice, V. 33, Issue 1 (1999). Available at http://www.sciencedirect.com/science/article/B6VG7-3VCC88F-3/2/2a0bc271ba272f8398e67867cda48ccd.

[12] Kimber, R. M. "The Traffic Capacity of Roundabouts." L.R. 942 (1980).

[13] Klar, Axel, Reinhart D. Kühne, and Raimund Wegener. "Mathematical Models for Vehicular Traffic." Surv. Math. Ind., 6 (1996). Available at http://citeseer.ist.psu.edu/old/518818.html.

[14] Nagel, Kai, Dietrich E. Wolf, Peter Wagner, and Patrice Simon. "Two-lane traffic rules for cellular automata: A systematic approach." Phys. Rev. E, V. 58 (1998). Available at http://prola.aps.org/pdf/PRE/v58/i2/p1425_1.

[15] Polus, Abishai and Sitvanit Shmueli. "Analysis and Evaluation of the Capacity of Roundabouts." Transportation Research Record: Journal of them Transportation Research Board, V. 1572 (1997). Available at http://trb.metapress.com/content/p1j1777227757852.

[16] Polus, Abishai, Sitvanit Shmueli Lazar and Moshe Livneh. "Critical Gap as a Function of Waiting Time in Determining Roundabout Capacity." J. Transp. Engrg. V. 129, Issue 5 (2003).

[17] Schreckenberg, M., A. Schadschneider, K. Nagel, and N. Ito. "Discrete Stochastic Models for Traffic Flow." Phys. Rev. E 51, V. 51 (1995). Available at

[18] Zein, Sany R., Erica Geddes, Suzanne Hemsing, and Mavis Johnson. "Safety Benefits of Traffic Calming." Transportation Research Record: Journal of the Transportation Research Board, V. 1578 (1997).

```java
package car;

import java.util.ArrayList;

public class AggressiveDriver implements Driver {

    private EntryLocation start;
    private ExitLocation end;
    private Location curr;
    private int speed;
    private int offset;
    private int totalTimePassed;
    private int roadTimePassed;
    private int yield;
    private boolean onRoad;

    public AggressiveDriver(EntryLocation start, ExitLocation end, int yield) {
        this.start = start;
        this.end = end;
        this.speed  = 2;
        this.curr = start;
        this.offset = (int)(Math.random() * speed);
        totalTimePassed = 0;
        roadTimePassed = 0;
        this.yield = yield;
        onRoad = false;
    }

    public ExitLocation getExit() {
        return end;
    }

    public ArrayList<Location> getNext(Road theRoad, ArrayList<Driver> drivers) {
        if (yield == 0) {
            ArrayList<Location> ret = new ArrayList<Location>();
            Location next = theRoad.getNext(curr);
            if (!next.isOccupied()) {
                if (curr.isEntry()) {
                    if (theRoad.getPrev(next).isOccupied()) {
                        next = null;
                    }
                }
            } else {
                next = null;
            }
            Location right = theRoad.getRight(curr);
            if (right != null && !right.isOccupied()) {
                if (theRoad.getPrev(right).isOccupied()) {
                    right = null;
                }
            } else {
                right = null;
            }
            Location left = theRoad.getLeft(curr);
            if (left != null && !left.isOccupied()) {
                if (theRoad.getPrev(left).isOccupied()) {
                    left = null;
                }
            } else {
```



```java
                left = null;
            }
            if (curr.isEntry()) {
                if (next != null) {
                    ret.add(next);
                }
            } else if (theRoad.vDist(curr, end) > 4 * theRoad.hDist(curr, end)) {
                if (next != null) {
                    ret.add(next);
                }
                if (left != null) {
                    ret.add(left);
                }
                if (right != null) {
                    ret.add(right);
                }
            } //else if (theRoad.vDist(curr, end) <= theRoad.hDist(curr, end)) {
            //  if (right != null) {
            //      ret.add(right);
            //  }
            //}
        else if (theRoad.vDist(curr, end) <= 4*theRoad.hDist(curr, end)) {
                if (right != null) {
                    ret.add(right);
                }
                if (next != null) {
                    ret.add(next);
                }
            }
            ret.add(curr);
            return ret;
        } else if (yield == 1) {
            ArrayList<Location> ret = new ArrayList<Location>();
            Location next = theRoad.getNext(curr);
            if (!next.isOccupied() && !theRoad.needToYield(curr)) {
                if (curr.isEntry()) {
                    if (theRoad.getPrev(next).isOccupied()) {
                        next = null;
                    }
                }
            } else {
                next = null;
            }
            Location right = theRoad.getRight(curr);
            if (right != null && !right.isOccupied() && !theRoad.needToYield(curr)) {
                if (theRoad.getPrev(right).isOccupied()) {
                    right = null;
                }
            } else {
                right = null;
            }
            Location left = theRoad.getLeft(curr);
            if (left != null && !left.isOccupied() && !theRoad.needToYield(curr)) {
                if (theRoad.getPrev(left).isOccupied()) {
                    left = null;
                }
            } else {
                left = null;
            }
```

```
    if (curr.isEntry()) {
        if (next != null) {
            ret.add(next);
        }
    } else if (theRoad.vDist(curr, end) > 2 * theRoad.hDist(curr, end)) {
        if (next != null) {
            ret.add(next);
        }
        if (left != null) {
            ret.add(left);
        }
        if (right != null) {
            ret.add(right);
        }
    } else if (theRoad.vDist(curr, end) <= 2*theRoad.hDist(curr, end)) {
        if (right != null) {
            ret.add(right);
        }
    } else if (theRoad.vDist(curr, end) <= 3*theRoad.hDist(curr, end)) {
        if (right != null) {
            ret.add(right);
        }
        if (next != null) {
            ret.add(next);
        }
    }
    ret.add(curr);
    return ret;
} else {
    ArrayList<Location> ret = new ArrayList<Location>();
    Location next = theRoad.getNext(curr);
    if (!next.isOccupied()) {
        if (curr.isEntry()) {
            if (theRoad.getPrev(next).isOccupied()) {
            //  next = null;
            }
        }
    } else {
        next = null;
    }
    Location right = theRoad.getRight(curr);
    if (right != null && !right.isOccupied()) {
        if (theRoad.getPrev(right).isOccupied()) {
            right = null;
        }
    } else {
        right = null;
    }
    Location left = theRoad.getLeft(curr);
    if (left != null && !left.isOccupied()) {
        if (theRoad.getPrev(left).isOccupied()) {
            left = null;
        }
    } else {
        left = null;
    }
    if (curr.isEntry()) {
        if (next != null) {
            ret.add(next);
```



```
            }
        } else if (theRoad.vDist(curr, end) > 4 * theRoad.hDist(curr, end)) {
            if (next != null) {
                ret.add(next);
            }
            if (left != null) {
                ret.add(left);
            }
            if (right != null) {
                ret.add(right);
            }
        } //else if (theRoad.vDist(curr, end) <= theRoad.hDist(curr, end)) {
        //  if (right != null) {
        //      ret.add(right);
        //  }
        //}
    else if (theRoad.vDist(curr, end) <= 4*theRoad.hDist(curr, end)) {
            if (right != null) {
                ret.add(right);
            }
            if (next != null) {
                ret.add(next);
            }
        }
        ret.add(curr);
        return ret;
    }
}

public int getSpeed() {
    return speed;
}

public Location getLoc() {
    return curr;
}

public boolean isMove() {
    totalTimePassed++;
    if (onRoad) {
        roadTimePassed++;
    }
    if (offset % speed == 0) {
        offset++;
        return true;
    } else {
        offset++;
        return false;
    }
}

public void moveTo(Location loc) {
    curr = loc;
}

public Location getLocation() {
    return curr;
}
```

```
    public String toString() {
        return "Driver at: " + curr.toString();
    }

    public int totalTimePassed() {
        return totalTimePassed;
    }

    public int roadTimePassed() {
        return roadTimePassed;
    }

    public int type() {
        return 2;
    }

    public void enterRoad() {
        onRoad = true;
    }
}
```

```java
package car;
import java.util.*;

public interface Driver {

    public ArrayList<Location> getNext(Road theRoad, ArrayList<Driver> drivers);

    public int getSpeed();

    public boolean isMove();

    public void moveTo(Location loc);

    public Location getLocation();

    public ExitLocation getExit();

    public int totalTimePassed();

    public int roadTimePassed();

    public int type();

    public void enterRoad();
}
```

```java
package car;

import java.util.ArrayList;

public class StandardDriver implements Driver {

    private EntryLocation start;
    private ExitLocation end;
    private Location curr;
    private int speed;
    private int offset;
    private int totalTimePassed;
    private int roadTimePassed;
    private int yield;
    private boolean onRoad;

    public StandardDriver(EntryLocation start, ExitLocation end, int yield) {
        this.start = start;
        this.end = end;
        this.speed  = 1;
        this.curr = start;
        this.offset = (int)(Math.random() * speed);
        totalTimePassed = 0;
        roadTimePassed = 0;
        this.yield = yield;
        onRoad = false;
    }

    public ExitLocation getExit() {
        return end;
    }

    public ArrayList<Location> getNext(Road theRoad, ArrayList<Driver> drivers) {
        if (yield == 0) {
            ArrayList<Location> ret = new ArrayList<Location>();
            Location next = theRoad.getNext(curr);
            if (!next.isOccupied()) {
                if (curr.isEntry()) {
                    if (theRoad.getPrev(next).isOccupied()) {
                        next = null;
                    }
                }
            } else {
                next = null;
            }
            Location right = theRoad.getRight(curr);
            if (right != null && !right.isOccupied()) {
                if (theRoad.getPrev(right).isOccupied()) {
                    right = null;
                }
            } else {
                right = null;
            }
            Location left = theRoad.getLeft(curr);
            if (left != null && !left.isOccupied()) {
                if (theRoad.getPrev(left).isOccupied()) {
                    left = null;
                }
            } else {
```



```java
                left = null;
            }
            if (curr.isEntry()) {
                if (next != null) {
                    ret.add(next);
                }
            } else if (theRoad.vDist(curr, end) > 4 * theRoad.hDist(curr, end)) {
                if (next != null) {
                    ret.add(next);
                }
                if (left != null) {
                    ret.add(left);
                }
                if (right != null) {
                    ret.add(right);
                }
            } //else if (theRoad.vDist(curr, end) <= theRoad.hDist(curr, end)) {
            //  if (right != null) {
            //      ret.add(right);
            //  }
            //}
        else if (theRoad.vDist(curr, end) <= 4*theRoad.hDist(curr, end)) {
                if (right != null) {
                    ret.add(right);
                }
                if (next != null) {
                    ret.add(next);
                }
            }
            ret.add(curr);
            return ret;
        } else if (yield == 1) {
            ArrayList<Location> ret = new ArrayList<Location>();
            Location next = theRoad.getNext(curr);
            if (!next.isOccupied() && !theRoad.needToYield(curr)) {
                if (curr.isEntry()) {
                    if (theRoad.getPrev(next).isOccupied()) {
                        next = null;
                    }
                }
            } else {
                next = null;
            }
            Location right = theRoad.getRight(curr);
            if (right != null && !right.isOccupied() && !theRoad.needToYield(curr)) {
                if (theRoad.getPrev(right).isOccupied()) {
                    right = null;
                }
            } else {
                right = null;
            }
            Location left = theRoad.getLeft(curr);
            if (left != null && !left.isOccupied() && !theRoad.needToYield(curr)) {
                if (theRoad.getPrev(left).isOccupied()) {
                    left = null;
                }
            } else {
                left = null;
            }
```

```
        if (curr.isEntry()) {
            if (next != null) {
                ret.add(next);
            }
        } else if (theRoad.vDist(curr, end) > 2 * theRoad.hDist(curr, end)) {
            if (next != null) {
                ret.add(next);
            }
            if (left != null) {
                ret.add(left);
            }
            if (right != null) {
                ret.add(right);
            }
        } else if (theRoad.vDist(curr, end) <= 2*theRoad.hDist(curr, end)) {
            if (right != null) {
                ret.add(right);
            }
        } else if (theRoad.vDist(curr, end) <= 3*theRoad.hDist(curr, end)) {
            if (right != null) {
                ret.add(right);
            }
            if (next != null) {
                ret.add(next);
            }
        }
        ret.add(curr);
        return ret;
    } else {
        ArrayList<Location> ret = new ArrayList<Location>();
        Location next = theRoad.getNext(curr);
        if (!next.isOccupied()) {
            if (curr.isEntry()) {
                if (theRoad.getPrev(next).isOccupied()) {
                //  next = null;
                }
            }
        } else {
            next = null;
        }
        Location right = theRoad.getRight(curr);
        if (right != null && !right.isOccupied()) {
            if (theRoad.getPrev(right).isOccupied()) {
                right = null;
            }
        } else {
            right = null;
        }
        Location left = theRoad.getLeft(curr);
        if (left != null && !left.isOccupied()) {
            if (theRoad.getPrev(left).isOccupied()) {
                left = null;
            }
        } else {
            left = null;
        }
        if (curr.isEntry()) {
            if (next != null) {
                ret.add(next);
```



```
            }
        } else if (theRoad.vDist(curr, end) > 4 * theRoad.hDist(curr, end)) {
            if (next != null) {
                ret.add(next);
            }
            if (left != null) {
                ret.add(left);
            }
            if (right != null) {
                ret.add(right);
            }
        } //else if (theRoad.vDist(curr, end) <= theRoad.hDist(curr, end)) {
        //  if (right != null) {
        //      ret.add(right);
        //  }
        //}
    else if (theRoad.vDist(curr, end) <= 4*theRoad.hDist(curr, end)) {
            if (right != null) {
                ret.add(right);
            }
            if (next != null) {
                ret.add(next);
            }
        }
        ret.add(curr);
        return ret;
    }
}

public int getSpeed() {
    return speed;
}

public Location getLoc() {
    return curr;
}

public boolean isMove() {
    totalTimePassed++;
    if (onRoad) {
        roadTimePassed++;
    }
    if (offset % speed == 0) {
        offset++;
        return true;
    } else {
        offset++;
        return false;
    }
}

public void moveTo(Location loc) {
    curr = loc;
}

public Location getLocation() {
    return curr;
}
```

```
    public String toString() {
        return "Driver at: " + curr.toString();
    }

    public int totalTimePassed() {
        return totalTimePassed;
    }

    public int roadTimePassed() {
        return roadTimePassed;
    }

    public int type() {
        return 1;
    }

    public void enterRoad() {
        onRoad = true;
    }
}
```

```java
package map;

import java.util.ArrayList;

public class EntryLocation extends Location {

    private int time;

    private ArrayList<Driver> waiting;

    public EntryLocation (int xPos, int yPos) {
        super(xPos, yPos);
        time = 0;
        waiting = new ArrayList<Driver>();
    }

    public boolean isEntry() {
        return true;
    }

    public void addDriver(Driver d) {
        waiting.add(d);
    }

    public Driver getDriver() {
        if (waiting.size() > 0) {
            Driver d = waiting.get(0);
            waiting.remove(0);
            return d;
        } else {
            return null;
        }
    }

    public ArrayList<Driver> waitlist() {
        return waiting;
    }
}
```

```java
package map;

public class ExitLocation extends Location {

    public ExitLocation(int xPos, int yPos) {
        super(xPos, yPos);
    }

    public boolean isExit() {
        return true;
    }
}
```

```java
package map;

public class Location {

    private int xPos;
    private int yPos;
    private boolean occupied;
    private int type;

    public Location(int xPos, int yPos) {
        this.xPos = xPos;
        this.yPos = yPos;
        this.occupied = false;
    }

    public int getXPos() {
        return xPos;
    }

    public int getYPos() {
        return yPos;
    }

    public boolean isOccupied() {
        return occupied;
    }

    public void flipOccupied() {
        occupied = !occupied;
    }

    public int type() {
        if (!occupied) {
            return -1; // not occupied
        } else {
            return type; // 1 for standard, 2 for aggressive
        }
    }

    public void setType(int type) {
        this.type = type;
    }

    public boolean isEntry() {
        return false;
    }

    public boolean isExit() {
        return false;
    }

    public String toString() {
        return "(" + xPos + " , " + yPos + ")";
    }
}
```

```java
package map;

import java.util.ArrayList;

public class Road {

    public Road(int width, int length, ArrayList<Integer> entries,
            ArrayList<Integer> exits)
    {
        this.width = width;
        this.length = length;
        road = new ArrayList<ArrayList<Location>>(width + 1);
        this.entries = new ArrayList<EntryLocation>(entries.size());
        this.exits = new ArrayList<ExitLocation>(exits.size());
        for (int i = 0; i < width + 1; i++)
        {
            road.add(new ArrayList<Location>(length));
            for (int j = 0; j < length; j++)
                road.get(i).add(new Location(i,j));
        }
        for (Integer i : entries)
        {
            EntryLocation entLoc = new EntryLocation(width, i.intValue());
            this.entries.add(entLoc);
            road.get(width).set(i.intValue(), entLoc);
        }

        for (Integer i : exits)
        {
            ExitLocation exitLoc = new ExitLocation(width - 1, i.intValue());
            this.exits.add(exitLoc);
            road.get(width - 1).set(i.intValue(), exitLoc);
        }
    }

    public Location getNext(Location current)
    {
        if (current.isEntry())
            return (road.get(width - 1).get(current.getYPos()));
        else
            return road.get(current.getXPos()).get((current.getYPos() + 1) % length);
    }

    public Location getLeft(Location current)
    {
        if (current.isEntry() || current.getXPos() == 0)
            return null;
        else
            return road.get(current.getXPos() - 1).get(current.getYPos());
    }

    public Location getRight(Location current)
    {
        if (current.isEntry() || current.getXPos() == width - 1)
            return null;
        else
            return road.get(current.getXPos() + 1).get(current.getYPos());
    }
```



```java
    public Location getPrev(Location current)
    {
        if (current.isEntry())
            return null;
        else
            return road.get(current.getXPos()).get((current.getYPos() - 1 + length)
                    % length);
    }

    public int getWidth()
    {
        return width;
    }

    public int getLength()
    {
        return length;
    }

    public ArrayList<EntryLocation> getEntries()
    {
        return entries;
    }

    public ArrayList<ExitLocation> getExits()
    {
        return exits;
    }

    public int vDist(Location from, Location to)
    {
            return ((to.getYPos() - from.getYPos() + length) % length);
    }

    public int hDist(Location from, Location to)
    {
        if (from.isEntry() || to.isEntry())
            return width;
        else
            return (to.getXPos() - from.getXPos());
    }

    public boolean needToYield(Location current)
    {
        if (current.getXPos() == width - 1)
        {
            Location loc = road.get(width).get((current.getYPos() + 1) % length);
            if (loc.isEntry() && loc.isOccupied())
                return true;
        }
        return false;
    }

    private ArrayList<ArrayList<Location>> road;
    private int width;
    private int length;
    private ArrayList<EntryLocation> entries;
    private ArrayList<ExitLocation> exits;
```

```
    public String toVisual() {
        String ret = "";
        for (int i = 0; i < width; i++) {
            for (int j = 0; j < length; j++){
                if (road.get(i).get(j).isOccupied()) {
                    ret += "S";
                }
                else {
                    ret += " ";
                }

            }
            ret += "\n";
        }

        for (int i = 0; i < length; i++) {
            if (road.get(width).get(i).isEntry()) {
                ret += "E";
            }
            else {
                ret += " ";
            }
        }
        return ret;
    }
}
```

```java
package map;

public class TrafficLight {

    public TrafficLight(Location roadLoc, EntryLocation entryLoc)
    {
        this.roadLoc = roadLoc;
        this.entryLoc = entryLoc;
        roadGo = true;
        entryGo = false;
    }

    public TrafficLight(Location roadLoc, EntryLocation entryLoc, boolean roadGo,
            boolean enterGo)
    {
        this(roadLoc, entryLoc);
        this.roadGo = roadGo;
        this.entryGo = enterGo;
    }

    public Location getRoadLoc()
    {
        return roadLoc;
    }

    public Location getEntryLoc()
    {
        return entryLoc;
    }

    public boolean getRoadGo()
    {
        return roadGo;
    }

    public boolean getEntryGo()
    {
        return entryGo;
    }

    public void flip()
    {
        roadGo = !roadGo;
        entryGo = !entryGo;
    }

    private boolean roadGo;
    private boolean entryGo;
    private Location roadLoc;
    private EntryLocation entryLoc;
}
```

```java
package simulator;

import java.util.ArrayList;

public class RandomTrafficLightController implements TrafficLightController{

    public RandomTrafficLightController(Road road,
            ArrayList<EntryLocation> entryLocs, int green, int red)
    {
        lights = new ArrayList<TrafficLight>(entryLocs.size());
        flipRed = new ArrayList<Integer>(entryLocs.size());
        flipGreen = new ArrayList<Integer>(entryLocs.size());
        greenPlusRed = green + red;

        Random generator = new Random();

        for (int i = 0; i < entryLocs.size(); i++)
        {
            int flipRedTime = generator.nextInt(greenPlusRed);
            flipRed.add(new Integer(flipRedTime));
            flipGreen.add(new Integer((flipRedTime + red) % greenPlusRed));
        }

        for (int i = 0; i < entryLocs.size(); i++)
        {
            boolean roadGo = generator.nextBoolean();
            lights.add(new TrafficLight(road.getPrev(road.getNext(entryLocs.get(i))),
                    entryLocs.get(i), roadGo, !roadGo));
        }
        time = -1;
        for (int i = 0; i < greenPlusRed; i++)
            this.step();
    }

    public HashSet<Location> step()
    {
        time++;
        for (int i = 0; i < greenPlusRed; i++)
        {
            if ((time % greenPlusRed) == flipRed.get(i).intValue() ||
                    (time % greenPlusRed) == flipGreen.get(i).intValue())
                lights.get(i).flip();
        }

        HashSet<Location> stoppedLocs = new HashSet<Location>(lights.size());
        for (TrafficLight light : lights)
        {
            if (!light.getRoadGo())
                stoppedLocs.add(light.getRoadLoc());
            else if(!light.getEntryGo())
                stoppedLocs.add(light.getEntryLoc());
        }
        return stoppedLocs;
    }

    public ArrayList<TrafficLight> getLights()
    {
        return lights;
    }
```



```java
    private ArrayList<TrafficLight> lights;
    private ArrayList<Integer> flipRed;
    private ArrayList<Integer> flipGreen;
    private int time;
    private int greenPlusRed;

}
```

```
package simulator;

import java.io.BufferedReader;

public class Runner {
    /**
     * @param args
     */
    public static void main(String[] args) {
        DecimalFormat formatter = new DecimalFormat("###.##");
        BufferedReader in = null;
        int numCases = 0;
        try {
            in = new BufferedReader(new FileReader("test.in"));
            numCases = Integer.parseInt(in.readLine());
        } catch (IOException e) {
        }

        for (int t = 0; t < numCases; t++) {

            int lanes = 0;
            int length = 0;
            int numRoads = 0;
            int isYield = 0;
            int lightType = 0;
            double genProb = 0;
            int green = 0;
            int red = 0;
            int mult = 0;

            try {
                String input = in.readLine();
                String[] inputs = input.split(" ");
                lanes = Integer.parseInt(inputs[0]);
                length = Integer.parseInt(inputs[1]);
                numRoads = Integer.parseInt(inputs[2]);
                isYield = Integer.parseInt(inputs[3]); // 0 for outer yield, 1 for inner
yield
                lightType = Integer.parseInt(inputs[4]); // 0 for none, 1 for uniform, 2
for synch, 3 for random
                genProb = Double.parseDouble(inputs[5]);

                if (lightType > 0) {
                    green = Integer.parseInt(inputs[6]);
                    red = Integer.parseInt(inputs[7]);
                }
                if (lightType == 2) {
                    mult = Integer.parseInt(inputs[8]);
                }
            } catch (IOException e) {
            }

            ArrayList<Integer> entrances = new ArrayList<Integer>();
            ArrayList<Integer> exits = new ArrayList<Integer>();
            for (int i = 0; i < numRoads; i++) {
                exits.add(i*length/numRoads);
                entrances.add(i*length/numRoads + 1);
            }
```



```
            int yield = isYield;

            Simulator theSimulator = null;

            if (lightType == 0) {
                theSimulator = new Simulator(lanes, length, entrances, exits, yield,
 genProb);
            } else if (lightType == 1 || lightType == 3) {
                theSimulator = new TrafficLightSimulator(lanes, length, entrances, exits,
 yield, genProb, lightType, green, red);
            } else if (lightType == 2) {
                theSimulator = new TrafficLightSimulator(lanes, length, entrances, exits,
 yield, genProb, lightType, green, red, mult);
            }

            int time = 0;
            while (time < 30000) {
                theSimulator.step();
                time++;
                double avg = 0;
                double roadAvg = 0;
                int count = 0;
                for (Integer i : theSimulator.getTotalTimes()) {
                    avg += i;
                    count++;
                }
                count = 0;
                for (Integer i : theSimulator.getRoadTimes()) {
                    roadAvg += i;
                    count++;
                }
                roadAvg /= (count+0.0);
                avg /= (count + 0.0);
                System.out.println("Total time: " + formatter.format(avg) + " | Road
Time: " + formatter.format(roadAvg) + " | Number: " + count);
                System.out.println("Step: " + time + " \n" +
 theSimulator.getRoad().toVisual());
                //System.out.println(theSimulator.displayTrafficLights());
                System.out.println("\n");
            }
            try {
                BufferedWriter out = new BufferedWriter(new FileWriter("test.out", true));
                double finalAvg = 0;
                double finalRoadAvg = 0;
                int finalCount = 0;
                for (Integer i : theSimulator.getTotalTimes()) {
                    finalAvg += i;
                    finalCount++;
                }
                finalCount = 0;
                for (Integer i : theSimulator.getRoadTimes()) {
                    finalRoadAvg += i;
                    finalCount++;
                }
                finalAvg /= (finalCount + 0.0);
                finalRoadAvg /= (finalCount + 0.0);

                out.write(lanes +  " " + length + " " + numRoads +
                        " " + genProb + " ");
```

```
                out.write(finalCount + " ");
                out.write(formatter.format(finalAvg) + " ");
                out.write(formatter.format(finalRoadAvg) + " ");
                out.write("\n");
                out.close();

                /*
                out.write("Lns: " + lanes +  " | Len: " + length + " | NumRds:  " +
numRoads +
                        " | Dens: " + genProb + " | Yield: ");
                if (isYield == 0) {
                    out.write("Outer ");
                } else {
                    out.write("Inner ");
                }
                out.write("| Light: ");
                if (lightType == 0) {
                    out.write("None ");
                } else if (lightType == 1) {
                    out.write("Uniform ");
                } else if (lightType == 2) {
                    out.write("Synchronized ");
                } else if (lightType == 3) {
                    out.write("Randomized ");
                }
                out.write("| Number: " + finalCount + " ");
                out.write("| Total Average: " + formatter.format(finalAvg));
                out.write(" | Road Average: " + formatter.format(finalRoadAvg));
                out.write("\n");
                out.close();
                */
            } catch (IOException e) {
            }
        }
    }
}
```

```java
package simulator;

import java.util.ArrayList;

public class Simulator {

    protected Road theRoad;
    protected ArrayList<Driver> drivers;
    protected int lanes;
    protected int length;
    protected ArrayList<EntryLocation> starts;
    protected ArrayList<ExitLocation> ends;
    protected ArrayList<Integer> totalTimes;
    protected ArrayList<Integer> roadTimes;
    protected int numEnds;
    protected int yield; // true for entering yield, false for inside yield
    protected final double GEN_PROB;

    public Simulator(int lanes, int length, ArrayList<Integer> entrances,
            ArrayList<Integer> exits, int yield, double genProb) {
        this.lanes = lanes;
        this.length = length;
        theRoad = new Road(lanes, length, entrances, exits);
        drivers = new ArrayList<Driver>();
        totalTimes = new ArrayList<Integer>();
        roadTimes = new ArrayList<Integer>();

        starts = theRoad.getEntries();
        ends = theRoad.getExits();
        numEnds = ends.size();
        for (EntryLocation e : starts) {
            int num = (int)(Math.random() * numEnds);
            drivers.add(new StandardDriver(e, ends.get(num), yield));
            e.setType(1);
            e.flipOccupied();
        }
        this.yield = yield;
        GEN_PROB = genProb;
    }

    public void step() {
        ArrayList<Driver> toRemove = new ArrayList<Driver>();
        ArrayList<Driver> toMove = new ArrayList<Driver>();
        for (Driver d : drivers) {
            if (d.isMove()) {
                toMove.add(d);
            }
        }

        int len = toMove.size();
        int num2;
        for (int i = 0; i < len; i++) {
            num2 = (int)(Math.random() * len);
            Driver temp = toMove.get(num2);
            toMove.remove(num2);
            toMove.add(temp);
        }
        for (Driver d : toMove) {
            ArrayList<Location> locs = d.getNext(theRoad, drivers);
```



```java
            if (!locs.isEmpty()) {
                if (locs.get(0) == d.getExit()) {
                    toRemove.add(d);
                }
                if (locs.get(0)!=d.getLocation()) {
                    d.enterRoad();
                    d.getLocation().flipOccupied();
                    d.moveTo(locs.get(0));
                    locs.get(0).flipOccupied();
                    locs.get(0).setType(d.type());
                }
            }
        }
        for (Driver d : toRemove) {
            d.getLocation().flipOccupied();
            totalTimes.add(d.totalTimePassed());
            roadTimes.add(d.roadTimePassed());
            drivers.remove(drivers.indexOf(d));
        }

        for (EntryLocation e : starts) {
            if (Math.random() > 1 - GEN_PROB) {
                int num = (int)(Math.random() * numEnds);
                int num3 = (int)(Math.random() * 2);
                if (num3==0 || num3==1) {
                    e.addDriver(new StandardDriver(e, ends.get(num), yield));
                } else {
                    e.addDriver(new AggressiveDriver(e, ends.get(num), yield));
                }
            }
        }

        for (EntryLocation e : starts) {
            for (Driver d : e.waitlist()) {
                d.isMove();
            }
            if (!e.isOccupied()) {
                Driver test = e.getDriver();
                if (test!= null) {
                    drivers.add(test);
                    e.flipOccupied();
                    e.setType(test.type());
                }
            }
        }
    }

    public Road getRoad() {
        return theRoad;
    }

    public ArrayList<Integer> getTotalTimes() {
        return totalTimes;
    }

    public ArrayList<Integer> getRoadTimes() {
        return roadTimes;
    }
```

```
    public String toString() {
        String ret = "Lanes: " + lanes + " | Length: " + length + " | ";
        for (Driver d : drivers) {
            ret += d.toString() + " | ";
        }
        return ret;
    }
//  Set<Location> forbidden = controller.step();
//  for (Driver d : drivers) {
//      if (d.isMove() && !forbidden.contains(d.getLocation())) {
//          toMove.add(d);
//      }

}
```

```java
package simulator;

import java.util.ArrayList;

public class SynchronizedTrafficLightController implements TrafficLightController {

    public SynchronizedTrafficLightController(Road road,
            ArrayList<EntryLocation> entryLocs, int green, int red, int multiplier)
    {
        lights = new ArrayList<TrafficLight>(entryLocs.size());
        flipRed = new ArrayList<Integer>(entryLocs.size());
        flipGreen = new ArrayList<Integer>(entryLocs.size());
        greenPlusRed = green + red;

        int flipRedTime = 0;
        for (int i = 0; i < entryLocs.size(); i++)
        {
            flipRed.add(new Integer(flipRedTime));
            flipGreen.add(new Integer((flipRedTime + red) % greenPlusRed));
            flipRedTime = (flipRedTime + road.vDist(entryLocs.get(i),
                    entryLocs.get((i+1)%entryLocs.size())) * multiplier) % greenPlusRed;
        }

        for (int i = 0; i < entryLocs.size(); i++)
        {
            boolean roadGo;
            if (flipRed.get(i).intValue() < flipGreen.get(i).intValue())
                roadGo = true;
            else
                roadGo = false;
            lights.add(new TrafficLight(road.getPrev(road.getNext(entryLocs.get(i))),
                    entryLocs.get(i), roadGo, !roadGo));
        }
        time = -1;
        for (int i = 0; i < greenPlusRed; i++)
            this.step();
    }

    public HashSet<Location> step()
    {
        time++;
        for (int i = 0; i < lights.size(); i++)
        {
            if ((time % greenPlusRed) == flipRed.get(i).intValue() ||
                    (time % greenPlusRed) == flipGreen.get(i).intValue())
                lights.get(i).flip();
        }

        HashSet<Location> stoppedLocs = new HashSet<Location>(lights.size());
        for (TrafficLight light : lights)
        {
            if (!light.getRoadGo())
                stoppedLocs.add(light.getRoadLoc());
            else if(!light.getEntryGo())
                stoppedLocs.add(light.getEntryLoc());
        }
        return stoppedLocs;
    }
```



```java
    public ArrayList<TrafficLight> getLights()
    {
        return lights;
    }

    private ArrayList<TrafficLight> lights;
    private ArrayList<Integer> flipRed;
    private ArrayList<Integer> flipGreen;
    private int time;
    private int greenPlusRed;
}
```

```java
package simulator;

import java.util.ArrayList;

public interface TrafficLightController {

    public HashSet<Location> step();

    public ArrayList<TrafficLight> getLights();

}
```

```java
package simulator;

import java.util.ArrayList;

public class TrafficLightSimulator extends Simulator {

    private TrafficLightController controller;

    public TrafficLightSimulator(int lanes, int length,
            ArrayList<Integer> entrances, ArrayList<Integer> exits,
            int yield, double genProb, int type, int green, int red) {
        super(lanes, length, entrances, exits, yield, genProb);
        if (type == 1) {
            controller = new UniformTrafficLightController(theRoad, starts, green, red);
        } else if (type == 3) {
            controller = new RandomTrafficLightController(theRoad, starts, green, red);
        }
    }

    public TrafficLightSimulator(int lanes, int length,
            ArrayList<Integer> entrances, ArrayList<Integer> exits,
            int yield, double genProb, int type, int green, int red, int mult) {
        super(lanes, length, entrances, exits, yield, genProb);
        if (type == 2) {
            controller = new SynchronizedTrafficLightController(theRoad, starts, green,
 red, mult);
        }
    }

    public void step() {
        ArrayList<Driver> toRemove = new ArrayList<Driver>();
        ArrayList<Driver> toMove = new ArrayList<Driver>();
        Set<Location> forbidden = controller.step();
        for (Driver d : drivers) {
            if (d.isMove() && !forbidden.contains(d.getLocation())) {
                toMove.add(d);
            }
        }

        int len = toMove.size();
        int num2;
        for (int i = 0; i < len; i++) {
            num2 = (int)(Math.random() * len);
            Driver temp = toMove.get(num2);
            toMove.remove(num2);
            toMove.add(temp);
        }
        for (Driver d : toMove) {
            ArrayList<Location> locs = d.getNext(theRoad, drivers);
            if (!locs.isEmpty()) {
                if (locs.get(0) == d.getExit()) {
                    toRemove.add(d);
                }
                if (locs.get(0)!=d.getLocation()) {
                    d.enterRoad();
                    d.getLocation().flipOccupied();
                    d.moveTo(locs.get(0));
                    locs.get(0).flipOccupied();
                    locs.get(0).setType(d.type());
```



```java
                }
            }
        }
        for (Driver d : toRemove) {
            d.getLocation().flipOccupied();
            totalTimes.add(d.totalTimePassed());
            roadTimes.add(d.roadTimePassed());
            drivers.remove(drivers.indexOf(d));
        }

        for (EntryLocation e : starts) {
            if (Math.random() > 1 - GEN_PROB) {
                int num = (int)(Math.random() * numEnds);
                int num3 = (int)(Math.random() * 2);
                if (num3==0 || num3==1) {
                    e.addDriver(new StandardDriver(e, ends.get(num), yield));
                } else {
                    e.addDriver(new AggressiveDriver(e, ends.get(num), yield));
                }
            }
        }

        for (EntryLocation e : starts) {
            if (!e.isOccupied()) {
                Driver test = e.getDriver();
                if (test!= null) {
                    drivers.add(test);
                    e.flipOccupied();
                    e.setType(test.type());
                }
            }
        }
    }

    public String toString() {
        String ret = "Lanes: " + lanes + " | Length: " + length + " | ";
        for (Driver d : drivers) {
            ret += d.toString() + " | ";
        }
        return ret + "\n" + displayTrafficLights();
    }

    public String displayTrafficLights() {
        String ret = " ";
        HashMap<Integer,Integer> lights = new HashMap<Integer,Integer>();
        for (int i = 0; i < controller.getLights().size(); i++) {
            lights.put(controller.getLights().get(i).getRoadLoc().getYPos(),i);
        }
        for (int i = 0; i < length; i++) {
            if (lights.containsKey(i)) {
                if (controller.getLights().get(lights.get(i)).getRoadGo()) {
                    ret += ">";
                }
                else if (controller.getLights().get(lights.get(i)).getEntryGo()){
                    ret += "^";
                }
            }
            else {
                ret += " ";
```

```
            }
        }
        return ret;
    }
}
```

```java
package simulator;

import java.util.ArrayList;

public class UniformTrafficLightController implements TrafficLightController {

    public UniformTrafficLightController(Road road, ArrayList<EntryLocation> entryLocs,
            int green, int red)
    {
        lights = new ArrayList<TrafficLight>(entryLocs.size());
        for (EntryLocation loc : entryLocs)
            lights.add(new TrafficLight(road.getPrev(road.getNext(loc)), loc));
        time = 0;
        this.green = green;
        this.red = red;
    }

    public HashSet<Location> step()
    {
        time++;
        if (time % (green + red) == 0 || time % (green + red) == green)
        {
            for (TrafficLight light : lights)
                light.flip();
        }

        HashSet<Location> stoppedLocs = new HashSet<Location>(lights.size());
        for (TrafficLight light : lights)
        {
            if (!light.getRoadGo())
                stoppedLocs.add(light.getRoadLoc());
            else if(!light.getEntryGo())
                stoppedLocs.add(light.getEntryLoc());
        }
        return stoppedLocs;
    }

    public ArrayList<TrafficLight> getLights()
    {
        return lights;
    }

    private int green;
    private int red;
    private ArrayList<TrafficLight> lights;
    private int time;
}
```