Traffic Signal Simulator
Main Approach

Introduction/Problem

The team is assigned the task to build an educational traffic flow simulation for Professor E's students in his/her traffic signal timing class because his/her students are having trouble grasping the theories that lies behind traffic signal timings. Professor E wants his/her students to be able to "play" with different traffic signal timing schemes to explore different scenarios. She anticipates that this will allow her students to learn from practice, by observing first-hand some of the traffic patterns that emerge under different signal timing conditions.

Essence/Core Problem

The essence of this design project how the traffic signal timing schemes and sequences are going to get implemented for each of the traffic lights on each intersections. Also, for the simulator to be as educational for the students as possible.

Traffic signal timing involves determining the amount of time that each of an intersection traffic lights spend being green, yellow, and red, in order to allow cars to flow through the intersection from each direction in a fluid manner. In the ideal case, the amount of time that people spend waiting is minimized by the chosen settings for a given intersection traffic lights. This can be a very subtle matter: changing the timing at a single intersection by a couple of seconds can have far-fetching effects on the traffic in the surrounding areas.

Professor E's main goal is for his/her students to be able to explore multiple alternative approaches. Students should be able to observe any problems with their map's timing scheme, alter it, and see the results of their changes on the traffic patterns.

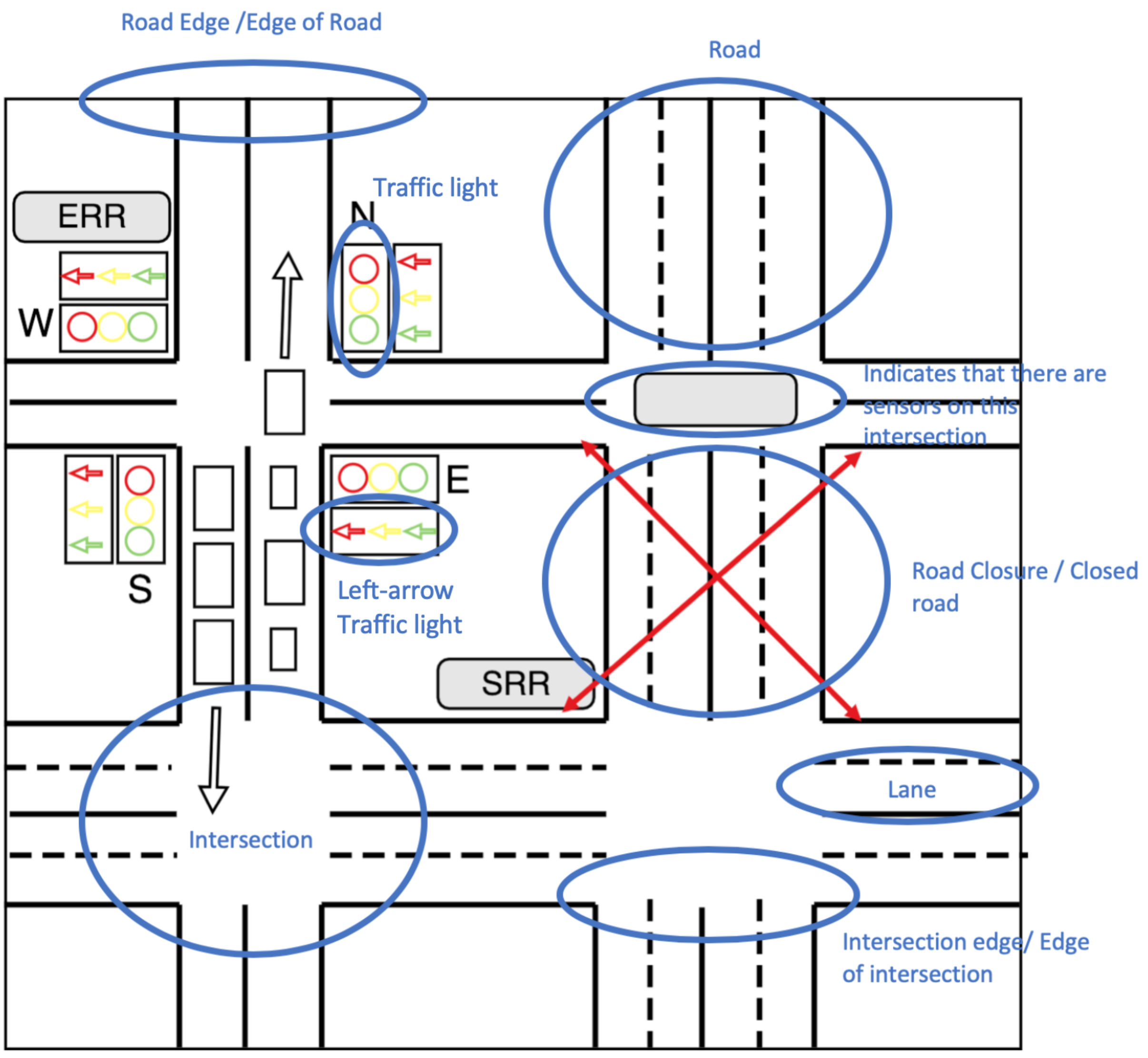
Desired Outcome

The team is to design the non-graphical portion of the traffic simulator program. We can assume that a different team will be responsible for the building User Interface for the simulator.

Our team must design the data structures and any associated APIs through which the program will represent the state of the simulation

Our team should design a basic algorithm that advances the state of the simulation one 'clock tick', that is, advances all elements of the simulation one step (e.g. move a car forward, change a light if it is necessary, have a new car enter the simulation)

Definition of Terms (Visually)


Road Edge /Edge of Road
Road
ERR
Traffic light
N
W
Indicates that there are sensors on this intersection
E
S
Left-arrow Traffic light
SRR
Road Closure / Closed road
Lane
Intersection
Intersection edge/ Edge of intersection

Requirements

1. Create Visual Map
    a. Can lay out roads in a pattern of the user's choosing
    b. Does not have to be complex
    c. Roads can be of varying length
    d. Different arrangement of intersections can be created
    e. Should accommodate at least 6 separate intersections
2. Students must be able to specify and adjust the behavior of the traffic lights at each of the intersections. A variety of sequences and timing schemes should be allowed. In addition:
    a. Should be able to accomodate left-hand turns protected by left-hand green arrow lights (that is, special lanes for turning left that have their own light governing whether traffic may or may not go)
    b. Combinations of traffic light states that would result in car crashes on an intersection are not allowed.

    c. Every intersection on the map must have traffic lights.
        i. No stop signs, overpasses, or other variations.
        ii. All intersections will be 4-way
            1. No “T” intersections
            2. No one-way roads
    d. Students must be able to design each intersection with or without the option to have sensors that detect whether any cars are present in a given lane.
        i. Students should be able to decide for themselves whether they want an intersection to have sensors or not.
3. Students must be able to simulate traffic flows on the map, based on the map created, and the intersection timing schemes
    a. Traffic levels should be conveyed visually in real-time, as they emerge in the simulation
    b. Current state of the intersection’s traffic lights should be shown and updated when they change.
    c. For example, you may choose to display individual cars, or to use a more abstract representation
4. Students should be able to change incoming traffic density by controlling how many cars enter the simulation at each on the *roads on the edge*
    a. Should be possible to create busy road, a seldom used road, and any variations in between
5. Should be able to run multiple simulations side-by-side, so students can see the effects of changed parameters.
    a. Students should also be allowed to save relevant simulations and retrieve them for later comparison
6. The applications should allow two types of traffic: cars and trucks.
    a. Trucks move slower than cars, meaning cars can exhibit passing behavior
7. For further realism, it should be possible to model road closures, in which case individual cars will adapt their driving pattern to reach their destination.
8. Application should be simple
    a. Only 1 simulation can run at a time
    b. All cars advance at the same speed
    c. No further complications should be introduced
        i. No pedestrians crossing the road, no bridges, no accidents, no road closures
    d. All roads should be either horizontal or vertical
        i. No diagonal or winding roads are allowed

Audience

- Civil engineering students at UCI
- Professor E

- “Others” that will build the user interface
- Application maintenance team

Other Stakeholders

- Drivers
- Students at other schools
- Other users outside of Professor E’s class that are also interested in traffic light signal timing schemes

Assumptions

- The software will run on desktop computers in school
- There will be a different UX team that will build the graphical portion of the simulator program
- The country that the traffic lights is modeled after is United States of America so all traffic lights will have a red, yellow and green light.

Goals

- Students understand traffic signal timing better
- Software is “educational”
- Students should be able to create roads and intersections
- Students should be able to control traffic flow by controlling the amount of cars entering from each edge road
- Students should be able to change the timing of the traffic lights
- Should be able to run on different desktop Operating Systems
- Software allows students to see the effect of timing changes on how long cars take to get through the intersections
- Coding level
  - Terms are easy to understand
- Allow at least six intersections on the map
- Intersections have the option for sensors to detect if cars are present
- Students are provided educational feedback on the simulation through some kind of statistics visualization

Constraints

- Multiple simulations need to run at the same time
- Trucks travel at a slower speed than Cars
- All roads must be straight roads; no diagonal or winding roads, just horizontal and vertical.
- No special circumstances such as car accidents, bridges, pedestrians, etc.
- Only 4-way intersections
- At least 6 intersections
- Traffic light states that would result in crashes are not allowed
- Students are not allowed to make changes while the simulation is running; they must stop it, change the settings, and start over

Decisions (Design Overview)

- Map is Matrix-based (like a grid)
    - 2-dimensional array
    - Roads, lanes, intersections will have their coordinates on the map
- Timing is round robin based (more details on “Timing and Sequencing Logic” section below)
- Which Traffic light goes first at an intersection (Sequencing)? (more details on “Timing and Sequencing Logic” section below)
    - Default - fixed sequencing by us
    - But users have the option to prioritize which traffic light can go green first
- Traffic light (more details on “Timing and Sequencing Logic” section below)
    - Red -> stop
    - Yellow -> once car sees light as yellow at the *edge of the intersection* (the closest cell to the stoplight), car will stop.
        - If a car passes the intersection while a traffic light is still green but the traffic light changes to yellow in the middle of that car passing the intersection, the car can keep on going.
    - Green -> go
- There will be protected left-hand green arrow lights at every intersection. (more details on its behavior below at ______ section)
- Right turn only on green light (that is, cars and trucks cannot turn right if the traffic light is red)
- Sensor (more details at “Sensor Logic” section below)
    - Only an indication that a sensor is present, users don't have to know what it is but they should see it’s impact on their simulation
    - A sensor is present at all 4 sides at the edge of an intersection
- Vehicle Specifics
    - Turning is allowed
    - Can change lanes
    - Speed is fixed
    - Car
        - Faster than trucks
        - Can pass trucks
    - Trucks
        - Have half the speed of cars
        - Cannot pass cars
- Passing and Lane Changing
    - Passing and lane changing are solely based on the position and destination of the car.
- Car Destination
    - Users can choose exit edges.
    - Cars exit to the farthest exit edge first but on a rotation if there are other exit edges specified.

    - Use Dijkstra’s Algorithm to find shortest path to an exit destination edge.
- Lanes
    - Users can choose how many lanes they want per road
        - If they say 1 lane, it’s automatically 1 lane going north and 1 lane going south
    - Left-turn lanes are automatically added independently of the number of lanes that the user specifies
- Road Closure
    - 1 road closure will be from an intersection edge to the other intersection edge of a given road
    - There cannot be a road closure at any road edge (overlapping a road edge)
- Simulation
    - 1 simulation will finish when the last car is out of the grid
    - Multiple simulations can run simultaneously but only a maximum of 4 can run simultaneously.
    - User can play, pause, replay and fast forward a simulation but they can't modify it in the middle of the simulation running
    - If they want to modify it will count as a new simulation and the simulation will restart
    - User can also skip the visual simulation and just jump to the feedback/statistics of the simulation at the end.
    - 1 tick will advance the simulation
        - Advances a car to the next cell
        - Increments the timer on the traffic lights
        - Etc
- Feedback/Statistics Information
    - Feedback will be per intersection and per whole Simulation (more details on “Feedback/Statistics” implementation explanation
    - Can save previous simulation feedbacks for comparison

IMPLEMENTATION LOGICS

Timing and Sequencing Logic

Our core scheduling implementation of our timing and sequencing logic will be Round Robin based. We chose Round Robin because it will give each traffic light on the intersection a chance to have a turn, for a certain amount of time, based on the sequence that the user specifies.

Traffic Light Logic

- Green light: User has the freedom to manipulate the amount of time a traffic light spends being green at each of all of the four sides of the intersection. For example, all four of the traffic lights at an intersection can have the same amount of time it will spend being green or each of the traffic lights can have different amount of times it will spend being green. We decided on this because this

freedom to choose the amount of time the traffic lights spend being green will give the students of Professor E the chance of exploring multiple alternative approaches, regardless of whether their chosen time settings are bad or good. It will give the students a chance to learn from their chosen timing settings.
    - The user will have the option to specify a default value for the amount of time spent green for all traffic lights in the whole simulation. This option is included to allow for users to start off with a “control” simulation where all traffic lights are allotted the same amount of time to spend green. The user could then modify lights in the next simulation and compare the results of both to see how traffic was effected. This also eases the user’s burden of having to manually input all timings.
- <u>Yellow light:</u> User has the freedom to manipulate the amount of time the traffic lights spend being yellow. Unlike how the green light works, the amount of the time all four of the traffic lights spend being yellow will be the same at a certain intersection.That amount of time will be specified by the user. We decided on this implementation because the essence of traffic timing is based on how long cars are allowed to go through an intersection (green light) and how long cars are stopped at an intersection (red light). It is not really focused around a light that is telling the drivers that it will soon go from green to red, and usually results in cars slowing down to prepare to stop.
- <u>Red Light:</u> The amount of time the traffic lights spend being red at an intersection will be solely based upon the amount of time allotted to the other traffic lights’ green and yellow duration, along with the sequence in which they are ordered.

Sequencing
- For each of the Round Robin options below, users will have the freedom to sequence/prioritize each of the four traffic lights at a given intersection. Meaning that the user can essentially choose which traffic light will go green first, then second, etc.
- The sequence can be as long as the user wants. For example, at an intersection, there will be a traffic light on the north side(N), south side(S), west side(W), and east side(E) and the sequence can be, e.g. [N,W,S,E] or [N,S,E,W] or [N,S,W,W,E,N], etc. This will be implemented using a circular buffer.

Strictly Round Robin Option
- Given a sequence of traffic lights (e.g. [N,W,S,E]), this implementation will go through each of the traffic lights strictly based on the order that they are in. From our example, the first traffic light that will turn green is the one that is on the North side of the given intersection. After the specified amount of time being green is reached, it will go through the yellow light then to the red light. Only then, the next traffic light in the sequence can turn to green, which in our example would be the one on the West side of the intersection. If the last traffic light(E) in the sequence has turned red, the next traffic light to turn green would be the first one

in the sequence(N) behaving like a circular buffer. This implementation will basically follow this algorithm until all cars are gone from the simulation.

- We decided to add this implementation to allow flexibility to our design in the event that turning is a feature that Professor E would like to see implemented in the future. This implementation is ideal for turning because everytime one of the traffic lights on the intersection is green, all of the other traffic lights will be red no matter what which would allow for a safe left turn every time. Also, using this implementation, no matter timing settings and sequence the students choose, a car crash would never occur.
- This implementation would not allow oncoming traffic from the opposite side to go at the same time as the traffic with the green light. For example, the North traffic light and the South traffic light cannot be green at the same time even if it is possible for both of them to be green and not result in any car crash because this algorithm will always strictly follow the given sequence. The Evolved Round Robin option below can solve this problem.

Evolved Round Robin Option

- Given a sequence of traffic lights (e.g. [N,W,S,E]), this implementation will go through each of the traffic lights based on the order that they are in (similar to the "Strictly Round Robin Option" from above). However, in addition to this, the current traffic light will check to see if the traffic light opposite to it can also be turned green for its allotted time. This will only occur if the allotted time for the green light on the opposite side is less than or equal to the current traffic light's green light time and a car crash will not result in doing so. From our example, the first traffic light that will turn green is the one that is on the North side of the given intersection. It will then check to see if the traffic light opposite of the one that is green can also go (S). If the time allotted for being green in the South traffic light is less than or equal to that of the North one, then the South traffic light will also turn green. Once the North traffic light has turned red, the next traffic light in the sequence (W) will turn green. Again this would check for the same requirements as the last traffic light, but this time in the East traffic light. And so on…
    - We decided less than or equal to because we only care about the current traffic light in the sequence since at that moment that's the traffic light that's being prioritized.
- We decided on this implementation because it adds a layer of realism to our design based on how actual, real-world traffic lights work.
- This implementation has some benefits and some drawbacks in the off-chance that Professor E wants turning to be in the simulation. The benefit would be that it would allow for unprotected left turning, even though the car may have to wait a while. The drawback would be that protected left turning is not covered by this implementation and the Evolved Round Robin implementation would degenerate into Strictly Round Robin (at least for the sides that allow protected turning and the side opposite to it).

Left-Arrow Traffic Light Logic

This traffic lights' timing and sequencing logic works exactly like the logic stated above. The only difference, if anything, is that both the normal traffic light and the left-arrow traffic light of one intersection side works dependently. Meaning that traffic lights (the normal traffic light and the left-arrow light for a given intersection edge) will be sequenced as a whole unit, not individually.

Strictly Round Robin Option

- The addition of the left-arrow does not have a significant effect on this option; the rules stated above still hold true. The only difference is that the current light that is green, also has its left-arrow light as green too. The light to advance to red last, will determine how long the other light in the sequence must wait to turn green. This next traffic light in the sequence (when able to) will turn green, along with its left-arrow light.

Evolved Round Robin Option

- The addition of the left-arrow does not have a significant effect on this option; the rules stated above still hold true. The only difference is that once the current traffic light has turned green, the next light to turn green is the current traffic light's left-arrow traffic light. The same rules still apply for each light, meaning that both the normal traffic light and the left-turn arrow light will check to see if the opposite traffic light (normal or left-arrow light, respectively) can also turn green. By opposite traffic light, it means that if it is the normal traffic light's turn to go green, it's going to check the opposite intersection side's normal traffic light and vice versa if it is the left-arrow traffic light's turn to turn green, it will check the opposite intersection side's left-arrow traffic light.

Sensor Logic

If the students decide that they want sensors at a specific intersection, sensors would automatically be put on all lanes on all four sides of the intersection where cars would typically be stopped at on a red light (the edge of the intersection where the cars would wait for the traffic light to turn green).

Strictly Round Robin

- If the students decide to put sensors on an intersection with the Strictly Round Robin scheme, the first traffic light to turn green is the one that will experience the most amount of cars passing through (the busiest road, the road edge with more cars). This is effectively bypassing the sequence set by the user. This will be determined by comparing the total cars coming from each of the Road Edges. Once the first traffic light's allotted green time is up, it will check to see if there are cars waiting at any of the other traffic lights. If there is no car waiting, the current traffic light will remain green until a car comes to any of the other traffic lights. If there is one traffic light that has at least one car waiting, the current traffic light will progress to red and the traffic light with at least the one car waiting will have its light turn green for the specified time. However, if two or more of the traffic lights have at least a car waiting, the intersection will default to the sequence given by the user until there is only one or no traffic lights with a car waiting.

Evolved Round Robin

- If the students decide to put sensors on an intersection with the Evolved Round Robin scheme, the first traffic light to turn green is the one that will experience the most amount of cars passing through (the busiest road, the road edge with more cars). However, unlike in the Strictly Round Robin implementation, the first traffic light will also check to see if the traffic light opposite from it can also be green (as long as a car crash will not result from it). If requirements are met, they will both be green for the time allotted to the first traffic light. The first traffic light will then check to see if the sensors on the adjacent traffic light lanes (the seldom road) have a car waiting. If there is no car waiting, both lights will continue to stay green until a car is present at the adjacent traffic lights. If there is a car waiting, the first traffic light will progress to red and the traffic light with the car waiting will get a green light (and it will also check conditions for opposite traffic light to also turn green). Once the traffic light on the seldom road has been green for the allotted time, both progress to red and the traffic lights on the busy road will turn green again (even if there are cars still waiting on the seldom road).
    - This implementation will give the students the chance to see what happens if the “seldom” road is actually not that seldom, just has a bit less traffic than the “busy” road. In this case the implementation of the use of sensors on Evolved Round Robin will just degenerate to an intersection without sensors that uses Evolved Round Robin and be pointless.

<u>Left-Arrow Traffic Light Sensor Logic</u>

Strictly Round Robin Option

- The incorporation of the left-arrow light does not have any impact on the timing or sequencing of the traffic lights depicted in the section Sensor Logic, Strictly Round Robin. The only thing new is that when a normal traffic light turns green, its left-turn arrow light also will turn green. All other logic in the section describes the behavior of this implementation.

Evolved Round Robin Option

- During the evolved round robin sequence, the lights will not only check the traffic light opposite of the busier road to see if there are cars waiting, but it will also check the left turn lane to see if there are cars waiting in it to turn. This check for left turn lanes will occur while the lights for either direction are green, for the left turn lanes that exist next to the lanes that have a green light. This allows the left turns on the sides that are already green to have priority over cars waiting on the sides of the intersection that are less busy. If cars exist in the left turn lanes on green sides, the next turn in the round robin would be allowing cars to turn left on those sides. If not, the lights will then do the normal check on the less busy street side, and turn green on that side if there are cars waiting.

- We decided to go with this implementation to keep it in line with the implementation that we already have, using the same logic that was used to check the lanes on the less busy streets on the intersection.

Passing and Lane Changing Logic

- Passing and lane changing are solely based on the position and destination of the car. First of all, only cars will have the ability to pass and they can pass both cars and trucks based on certain conditions(explained in more detail below). Also, all cars will move at the same speed by default (implementation-wise, this means that a car will move 1 cell per 1 clock tick) but cars will have the ability to adapt to a truck's speed if necessary(explained below). However, though all trucks will also move at the same speed, trucks will move slower than cars (implementation-wise, this means that a truck will move 1 cell per 2 clock ticks).
- Essentially, there will be 2 reasons for passing to happen. The first reason is if a car is behind a truck. This is so that all cars behind a truck won't get backed up and have to take on the "speed" of a truck for the remainder of the simulation or until the truck turns (moving out of the way of the cars). The second reason is if a specific car (denoted as "this car") has a car in front of it with the speed of a truck. A car that is behind a truck and unable to pass it, will take on the speed of the truck's car. Likewise, a car behind the car stuck will also take on the speed of the car that is stuck (meaning both cars now have the same "speed" of the truck). This is so that if there are multiple cars stuck behind a truck, "this car" (which is coming up on this line of cars stuck behind a truck) can know that these cars are being slowed by a truck, and effectively change lanes to not get stuck behind the truck like the others. Once a car is able to change lanes and move from behind the truck, the car's speed goes back to its original fixed speed.
- If any of the 2 scenarios above are confronted by a car, the car will first have to check a few things before it can decide to pass. The first thing it should check is if it has time to make a passing maneuver. For example, the car should check in its path set that it is not expected to make a turn within the next 5 cells (effectively allowing the car to have time to pass the truck and move back into the lane it changed from before expecting to make a turn). If this is possible, the car should then check that the cell next to it (right and/or left depending on which one is better suited for the car) and the cell diagonal to it (right and/or left) to make sure there are no vehicles in the way to block the car from making a passing maneuver. It is also necessary to make sure that the car checks the direction of the lane it is wanting to change into (to prevent the car from ending up on the wrong side of the road) and to check to make sure it is not currently in an intersection (to prevent the cars from making an illegal move). If the car is not expecting to turn and there are no

cars obstructing the car to make the maneuver, then the car may move diagonally to the right or left one cell.

- Lane Changing is a very similar concept with minor modifications except that the scenario necessary to necessitate a lane change is different and can be both performed by a car or a truck. The reason for a lane change is that if the vehicle's path set contains a change in direction within the next 5 cells, the vehicle should check to see if it can change lanes (same check system as for the passing maneuver, except for the difference in what the vehicle is looking for in its path set). The vehicle will keep lane changing until it is in the leftmost/rightmost lane (with respect to which way they are turning).

Car Destination Logic

- Since the user can choose where to spawn vehicles, they can also pick the road edges to be end destinations for vehicles. The vehicles will proceed to the end destination using the road of shortest distance. The road of shortest distance will be calculated using Dijkstra's algorithm. Dijkstra's algorithm is used to figure out the shortest distance between two nodes (and in this case, two road edges).
- The spawned vehicles from a specified road edge will be dispersed on a rotation to the specified exit edges until all the vehicles have exited the simulation. This rotation sequence will start with the destination (road edge selected to be an exit) that is farthest from the given road edge (spawning the vehicles) and make its way to the closest road edge (exit). Each vehicle going to a specific road edge, will have the same path (set of directions, which is determined using Dijkstra's algorithm) as all other vehicles headed to the destination via that specific road edge. This was decided on, so that multiple iterations of a single simulation will yield the same results.
- Road edges can both spawn vehicles and be a destination for vehicles. However, a road edge that is both spawning vehicles and is a destination, will not be permitted to make the destination of its vehicles the same road edge. It will essentially exit to the next exit edge in line.

Simulation Logic

- All simulations are essentially named and no two simulations can have the same name (e.g. "Simulation 1", "Sim 2", "Best Simulation" etc…). For every new simulation (simulation with different inputs), it's inputs can be saved for later retrieval if the user wants to replay a past simulation. Our implementation will allow users to choose from a list of past simulations to re-run if they want to.

Feedback/Statistics Information

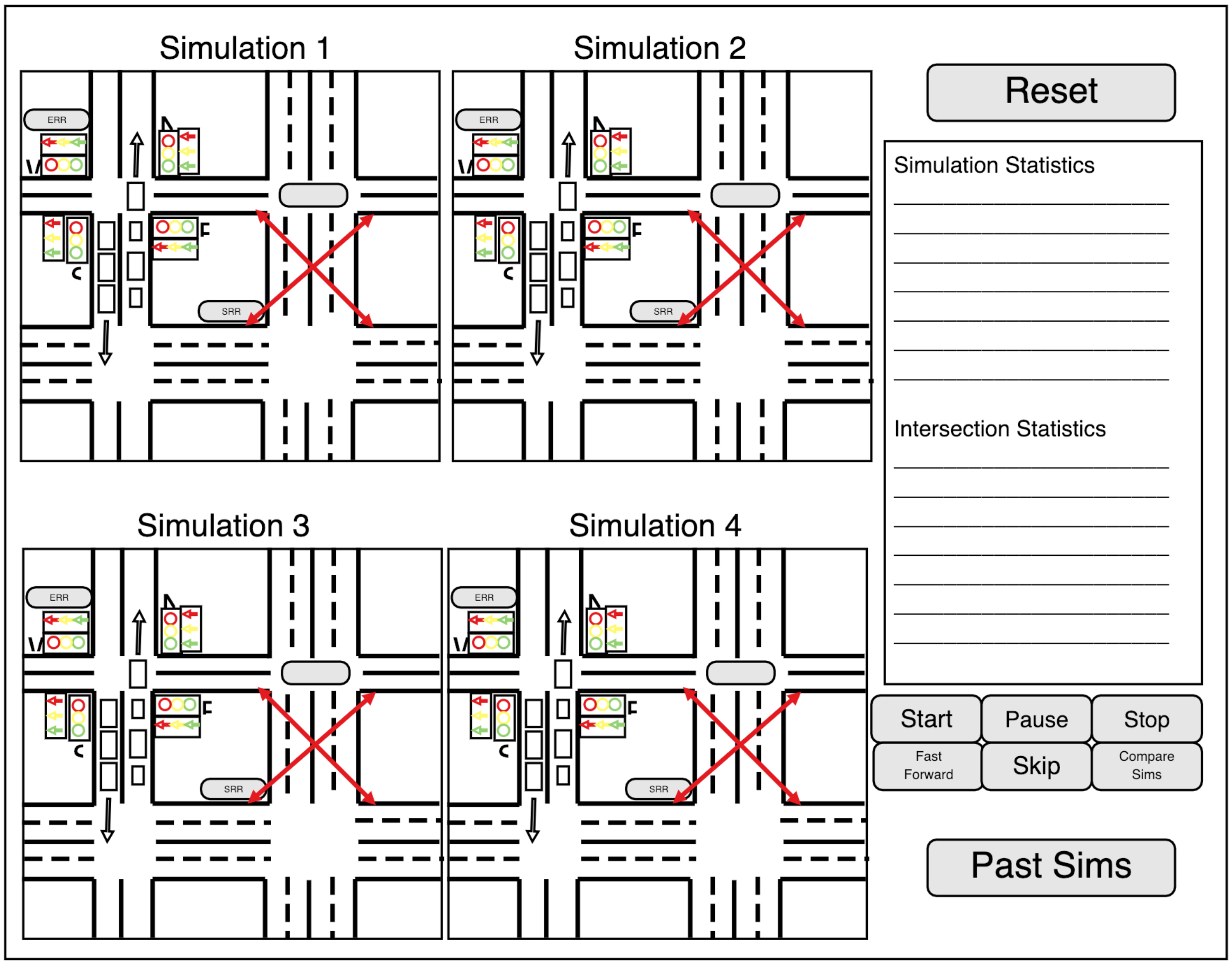
Statistical information will be derived from class attributes.

- Per Intersection
  - Average red light waiting time
  - Number of cars that passed the intersection
  - Number of lanes on each edge of the intersection

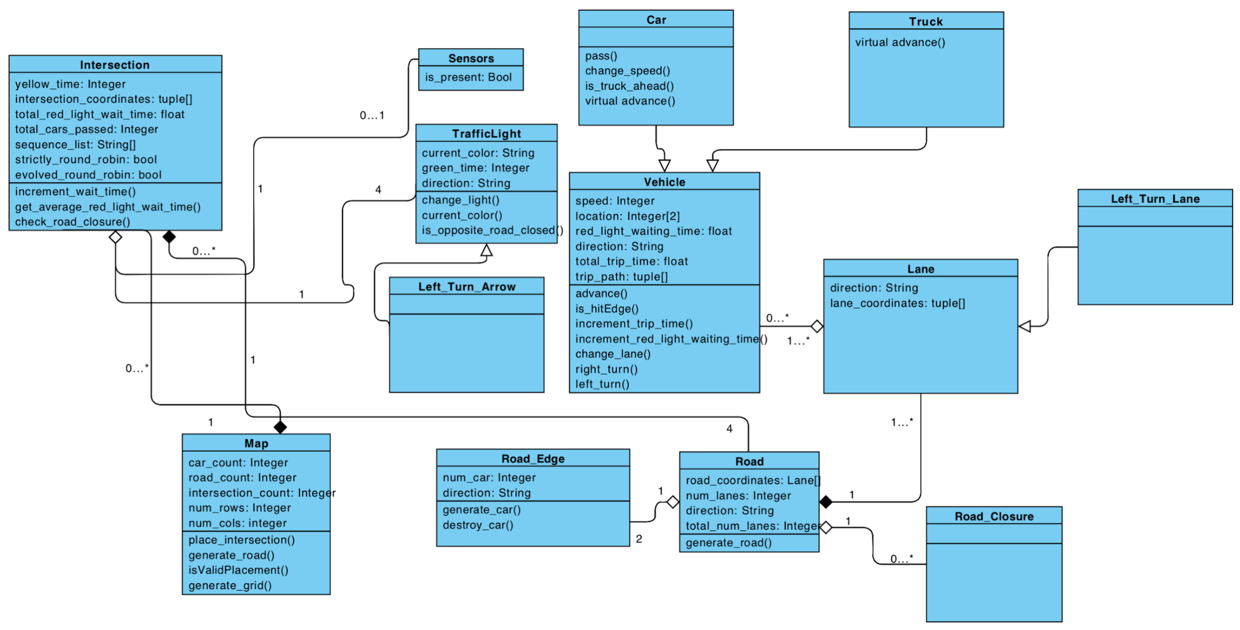
    - Green time setting for each of the traffic light at the intersection
    - Yellow time setting for the intersection
    - Indicate if there was a sensor present

- Simulation Statistics
    - Average time car spent in the simulation
    - Total number of cars in the whole simulation
    - Total numbers of intersections
    - Average red light waiting time
    - How long the simulation took
    - History of changes (compare simulations, doesn't have to be extensive but it's there)

Mock-Up



UML DIAGRAM



UML Diagram Explanation

Vehicle

- Attributes
  - speed: Integer
    - The car or the trucks current “speed”.
  - location: Integer[2]
    - The coordinates of where the car is currently on the grid
  - red_light_waiting_time: float
    - This will be helpful for the statistical feedback for the simulation
  - direction: String
    - This will be helpful on knowing how to advance the car one cell forward
  - total_trip_time: float
    - This will be helpful for the statistical feedback for the simulation
  - trip_path: tuple[]
    - Uses Dijkstra’s algorithm to find the shortest path to a destination/exit edge
- Methods
  - advance()
    - The method used to advance a car forward, with respect to its given direction
  - is_hitEdge() -> bool
    - The method used to determine if the car object is at a road edge and is exiting the map
  - increment_trip_time()

        - Increments total_trip_time attribute for the simulation statistics
    - increment_red_light_waiting_time()
        - Increments red_light_waiting_time attribute for the simulation statistics
    - change_lane()
        - Moves car from one lane to another. Has a set of conditions whether to change lane on the right or the left
    - right_turn()
        - Moves a vehicle in a way that will make it turn right based on a set of coordinates that will be derived from the car's current position
    - left_turn()
        - Moves a vehicle in a way that will make it turn left based on a set of coordinates that will be derived from the car's current position

Car

- Methods
    - pass()
        - Passes a car or a truck that has the truck's speed once certain conditions are met.
    - change_speed()
        - Changes car's speed accordingly
    - is_truck_ahead()
        - Checks if there is a car ahead. This will be useful for passing behavior
    - virtual_advance()
        - Car will have its own advance method. Overrides the Vehicle class' advance method since cars advance on a different speed than trucks

Truck

- Methods
    - virtual_advance()
        - Truck will have its own advance method. Overrides the Vehicle class' advance method since trucks advance on a different speed than cars

Lane

- Attributes
    - direction: String
        - This will be helpful in knowing which way cars are going while on these lanes and for other objects to be aware of which lanes provide oncoming traffic and which ones do not
    - lane_coordinates: tuple[]
        - A list of tuples that contain the coordinates of the entirety of the lane object

Left_Turn_Lane

- A child class of Lane object. This class indicates whether a lane is a left turn lane or not. We chose to make a child class from lane instead of just putting a boolean attribute in the Lane object to indicate a left turn lane because we think that this would be more understandable visually.

Road

- Attributes
    - road_coordinates: Lane[]
        - A list of lanes
    - total_num_lanes: Integer
        - Number of lanes in a given road object (counting lanes going North and going South)
    - Num_lanes: Integer
        - If the user specifies 1, 1 lane will be created for cars going North and 1 lane will be made for cars going south
    - direction: String
        - This will be helpful in knowing which way lanes within the road object are travelling
- Method
    - generate_road()
        - Based on num_lanes, lane objects will automatically be created to make a road

Road_Closure

- A child class of Road object. This class indicates whether a road is closed or not. We chose to make a child class from lane instead of just putting a boolean attribute in the Road object to indicate a closed road because we think that this would be more understandable visually.

Road_Edge

- Attributes
    - num_car: Integer
        - Number of cars that will emerge from this edge
    - total_get_through_time: float
        - The total amount of time that each car spent trying to get from the opposite road edge to this road edge
    - Direction: String
        - This will be helpful when creating car objects and passing down its direction to the cars. Also helpful in differentiating a car on the edge that was just created and one that needs to be destroyed
- Methods
    - generate_car()
        - Will keep on generating cars until it equals the num_car attribute
    - destroy_car()

        - Destroys a car object to free up memory and prevent it from segmentation faulting. Also takes the needed car statistics from it before destroying.
    - is_all_destroyed() -> bool
        - Checks if all the cars generated on this road edge have been destroyed
    - increment_total_get_through_time()
        - Adds a soon to be destroyed car's stats to the total_get_through_time attribute
    - get_average_get_through_time() -> float
        - Total_get_through_time divided by the number of cars on this road edge
        - This will be helpful for statistical feedback at the end of the simulation

Map

- Attributes
    - car_count: Integer
        - Contains the total number of cars in the simulation. Gets incremented with each new car created
    - road_count: Integer
        - Contains the total number of roads in the simulation
    - intersection_count: Integer
        - Contains the total number of intersections in the simulation.
    - num_rows: Integer
        - The Map object's number of rows. Methods to this are private, only for UX to understand the dimensions and not to be manipulated by a user
    - num_cols: Integer
        - The Map object's number of columns. Methods to this are private, only for UX to understand the dimensions and not to be manipulated by the user
- Methods
    - place_intersection()
        - In the grid, the user can click places to put intersections then generate intersection objects on the coordinates that they choose
        - When two automatically generated roads intersect, the overlapping coordinates become a new intersection by the system calling this method without the user even specifying that this should be an intersection
        - With this the map can accommodate at least 2 intersections but it doesn't have to start with at least 6 intersections
    - generate_road()
        - Generates roads extending from all four sides of the Intersection object to the edge of the Map object

        - is_valid_placement() -> bool
            - Checks to make sure that the intersection trying to be placed is in a valid place, e.g. not on top of an already existing intersection and not partially off the edge of the map
        - generate_grid()
            - Generates a grid based on the given number of rows and columns

Intersection

- Attributes
    - Yellow_time: Integer
        - The time allotted for traffic lights within the intersection to be yellow for
    - Intersection_coordinates: tuple[]
        - A list of tuples that contain the coordinates of the entirety of the intersection object
    - Total_red_light_wait_time: float
        - Total time that cars spent waiting at a red light at this intersection
    - Total_cars_passed: Integer
        - The number of cars that passed through the intersection
    - Sequence_list: String[]
        - The traffic light sequence is stored here and can be used by the intersection to guide the next traffic light to turn green (when necessary)
    - Strictly_round_robin: bool
        - This attribute is set to true on default and is the default option for sequencing
    - Evolved_round_robin: bool
        - This attribute is initially set to false, but can be chosen by the user. Once set by the (setter method) it will also turn the Strictly_round_robin attribute to false
- Methods
    - increment_wait_time()
        - Method that adds the waiting car’s time at the red light to the intersection’s total_red_light_wait_time.
    - get_average_red_light_wait_time()
        - Total_red_light_wait_time divided by the number of that passed through the intersection
        - This will be helpful for statistical feedback at the end of the simulation
    - check_road_closure()
        - Checks if a certain road connected to an intersection edge is closed or not

TrafficLight

- Attributes
    - current_color: String
        - The traffic light’s current state. From this the UX team that will design the graphical portion of the software can know which light is currently lit and which ones are not
    - green_time: Integer
        - The amount of the that this particular traffic light spends time being green
    - direction: String
        - Indicates which side of the intersection this traffic light is on. Inputs can be “N”, “S”,”E”, or ”W” for North, South, East, West
- Methods
    - change_light()
        - Method that changes the traffic light’s attribute current_color to the next color
    - current_color()
        - Method that gets the current color of the traffic light. Can be used later by UX to display the color of each traffic light at each intersection
    - is_opposite_road_closed()
        - Checks if the road opposite to the position of this traffic light is closed or not. If it is closed, this traffic light will just stay on the red light the whole time and it will skipped on the sequencing of that intersection’s traffic light timing scheme

Left_Turn_Arrow

- A child class of TrafficLight object. This class indicates whether a traffic light is the protected left-arrow traffic light or not. We chose to make a child class from lane instead of just putting a boolean attribute in the TrafficLight object to indicate a protected left-arrow traffic light because we think that this would be more understandable visually.

Sensors

- Attributes
    - is_present: bool
        - Method that indicates if a certain intersection has sensors

PSEUDOCODE/ALGORITHM that Advances the Simulation

```
Choose which traffic lights are turned green, if there is not already one green at each intersection (based off of sequence or busiest road implementation)
Check all traffic lights to see whether if they are to stay green, turn yellow, or turn red
    For Strictly Round Robin without Sensors
```

```
        Check to make sure that the current traffic light that is green has been green for less than its allotted time.
            If true, keep normal traffic light green and make sure left-turn light is also green
            If false, change normal traffic light to yellow along with its associated left-turn light
        Check to make sure that the current traffic light that is yellow has been yellow for less than its allotted time.
            If true, keep traffic light yellow
            If false, change both normal and left-turning traffic lights to red and turn the next traffic light(both normal and left-turn) in the sequence to green
    For Evolved Round Robin without Sensors
        Check to make sure that the current traffic light that is green has been green for less than its allotted time.
            Check if normal traffic light is green
                If true, keep normal traffic light green
                If false, change normal traffic light to yellow and its opposite if it is also green
            Else check if left-turning light is green
                If true, keep left-turning traffic light green
                If false, change left-turn traffic light (and its opposite if it is also green) to yellow
        Check to make sure that the current traffic light that is yellow has been yellow for less than its allotted time.
            Check if normal traffic light is yellow
                If true, keep normal traffic light yellow
                If false, change normal traffic light (and its opposite if it is also yellow) to red and turn the associated left-turn traffic light to green
            Else check if left-turning light is yellow
                If true, keep left-turning light yellow
                If false, change left-turning light( and its opposite if it is also yellow) to red and turn the next normal traffic light in the sequence to green
    For Strictly Round Robin with Sensors
        Check to make sure that the current traffic light that is green has been green for less than its allotted time.
            If true, keep normal traffic light green and make sure left-turn light is also green
            If false, check sensors on the other intersections
                If no car waiting, keep both normal and left-turn traffic lights green
                If there is a car waiting, change both normal and left-turn traffic lights to yellow
        Check to make sure that the current traffic light that is yellow has been yellow for less than its allotted time.
            If true,  keep normal traffic light yellow and make sure left-turn light is also yellow
            If false, changes both normal and left-turning traffic lights to red and checks sensors on the other intersections
                If only one sensor has a car waiting, both normal and left-turning traffic lights paired with the sensors turn green
                If 2 or more sensors have a car waiting, the intersection defaults to the traffic light sequence and chooses the next pair of normal and left-turning traffic lights to turn green based off of that
    For Evolved Round Robin with Sensors
        Check to make sure that the current traffic light that is green has been green for less than its allotted time.
            Check if normal traffic light is green
                If true, keep normal traffic light green
                If false, check sensors on the other intersections and the sensors of the current and opposite left-turning lanes
                    If no car waiting, keep normal traffic light green
                    If there is a car waiting, change normal traffic light (and its opposite if it is also green) to yellow
            Else check if left-turning light is green
                If true, keep left-turning traffic light green
                If false, check sensors on the other intersections and the sensors of the current and opposite non-left-turning lanes
                    If no car waiting, keep left-turning traffic light green
                    If there is a car waiting, change left-turning traffic light (and its opposite if it is also green) to yellow
        Check to make sure that the current traffic light that is yellow has been yellow for less than its allotted time.
            Check if normal traffic light is yellow
                If true, keep normal traffic light yellow
                If false, changes normal traffic light (and its opposite if yellow) to red and checks sensors on the other intersections and the sensors of the current and opposite left-turning lanes
                  If there are cars waiting at both adjacent traffic lights and at the current or opposite left-turning lanes, the intersection will default to following the traffic light sequence, however, giving priority to the left-turning lanes and first turning them green before moving to the adjacent traffic lights
                    Else if there is only a car waiting at a left-turning lane either on the current or opposite lane, the left-turning light of the current will turn green
                    Else if there is only a car waiting at an adjacent traffic light, then the traffic light with the waiting car will turn green
            Check if left-turning light is yellow
```

```
                If true, keep left-turning light yellow
                If false, changes left-turning traffic light (and its opposite if yellow) to red and checks sensors on the other intersections and the sensors of the current and opposite non-left-turning lanes
                    If there are cars waiting at both adjacent traffic lights and at the current or opposite non-left-turning lanes, the intersection will default to following the traffic light sequence, however, giving priority to the adjacent traffic lights and first turning one of them green before moving to the current and opposite non-left-turning lights
                    Else if there is only a car waiting at a non-left-turning lane either on the current or opposite lane, the non- left-turning light of the current will turn green
                    Else if there is only a car waiting at an adjacent traffic light, then the traffic light with the waiting car will turn green
For Intersections that have Evolved Round Robin with and without Sensors, check requirements for opposite traffic light
    If requirements met and normal traffic light is green, turn opposite normal traffic light green as well
    If requirements met and left-turn traffic light is green, turn opposite left-turn traffic light green as well
Destroy all cars that have reached their destination (their predetermined exit)
Advance all vehicles in the simulation by one cell if possible, starting with the vehicles nearest to the road edge with a direction opposite to that of the car's direction.
    Vehicles going straight should move forward, vehicles turning should move with respect to the turn needed to be made
    Do not advance: vehicle is at a red light; green light but there is a vehicle in the intersection that is not going the same direction( car caught in the middle of an intersection ); there is another vehicle object occupying the cell already
    Check to see if vehicle is in the correct lane (or needs to be in a specific lane)
        If not in correct lane, attempt to make a lane change
    If vehicle being advanced is a car and there is another vehicle occupying the cell in front
        Check if vehicle occupying the cell in front is a truck
            If true, check conditions for passing
                If conditions pass, perform passing maneuver
        Check if vehicle occupying the cell in front is a car with speed of a truck
            If true, check conditions for passing
                If conditions pass, perform passing maneuver
Create cars on Road Edge if necessary
Update the stats of all stat keeping attributes
~~Simulation has advanced One Clock Tick~~
```

REFLECTION

Main Strengths

- The biggest strength that our first implementation had was the fact that our timing and sequencing logic was extremely flexible. When the protected left-turn traffic lights were introduced, it did not destruct the flow of our first implementation as much because the left-turn traffic lights can still follow the same logic of our first implementation traffic lights. It's just instead of only having 4 traffic lights per intersection, there are now 8 traffic lights per intersection.
    - If anything the only modification that needed to be added was to our timing scheme for traffic lights to accommodate protected left turns. One of our timing schemes was a "Round Robin" so we had to integrate a turn for turning lights only, since those cannot be green when there is any oncoming traffic from 3 of the 4 directions. For lights with sensors, in addition to checking for waiting traffic on the less busy road, we also check for cars that want to turn left from the busier road.
    - All in all, the fact that our timing and sequencing logic still held true even with the added protected left turn traffic lights was a big plus to us because that tells us that we tackled the essence of this design project very well.
- Another big strength that our first implementation had was making our map grid/matrix-based. By making our map grid/matrix-based, even if we did not implement turning or lane changes in our first implementation, and even if adding turning and lane changing had high complexity, it was not that hard to implement when turning was added to the requirements because we just get the right coordinates from grid and have our vehicles move on those coordinates. We didn't have to redesign the whole map because making our map grid-based gave us flexibility for turning, lane changing, and even passing.
    - Changing lanes worked well because of the fact that our lanes already used a grid system. This allowed us to easily use a checking system to see whether or not a car existed in the grid adjacent to it for allowing lane changes.
    - This grid implementation also made finding the shortest path for Dijkstra's algorithm easier.
- Our original design already implemented cars checking to see if the grid in front of it contained a moving car. This allowed us to easily integrate trucks into the system.
- Trucks stay in one cell for two ticks, which is the way that we made them appear to move slower in our system. Having the traffic move at the same speed behind them was natural because the cars would already only move if the vehicle in front of it was moving. Therefore, the cars stuck behind a truck would never move through the truck, and this required very little extra adjustment on our part.
    - Our original design is limited in options for speed. If we wanted some cars to simulate 30 mph and others to simulate 38 mph, for example, it would not work well. Since we decided to have cars go exactly twice as fast as trucks, our original design still works with only a few modifications.
- Our road edges already created and destroyed cars to show cars entering and leaving the grid. This was helpful in our second design, since we had to spawn a certain amount of cars from each road edge, and the edges were considered the destinations as well. These edges already had counters for how many cars were both spawned and

destroyed throughout the simulation; we merely had to give it an initial amount of cars to make and have it count up to that number of cars using the existing functions.

- We anticipated the user wanting to compare statistics from previous simulations and already included summary statistics in our previous design. The only part we needed to add was the option to save and replay past simulations, which was straightforward because of the way we had structured the simulation data.

Main Weaknesses

- We needed to modify our UML to accommodate different vehicle types by creating a Vehicle parent class with Car and Truck subclasses. Similarly, we created a Lane parent and Left Turn Lane subclass, a Road parent and Closed Road subclass, and Traffic Light parent and Left Traffic Light subclass.
    - We didn’t anticipate enough that different types of vehicles may be added that’s why we didn't have a “Vehicle” parent class in our first implementation. This is a weakness because we had to essentially make our first implementation of “Car” to be “Vehicle” since cars and trucks behave differently
    - We also did not anticipate enough that a different type of traffic light may be introduced and though this is a form of weakness, the addition of the left-turn traffic light wasn’t too much of a downfall for our first implementation because as we mentioned above, the left-turn traffic light easily adapted to our timing and sequencing logic from our first implementation
- Though we mentioned above that adding turning was not essentially that hard to add because of the flexibility that our grid-based map gave us, not having the idea of turning in our first implementation, at all, was still a big weakness because of road closures. All cars only went straight in our first implementation and had no destination, or logic to get there, programmed into them. In addition, we had to add left turn lanes which were completely nonexistent at first. Once cars needed to deviate from their original path or turn, under the original program they would have no way to find their way to their original destination. We added a destination attribute to each car and gave the cars a routing algorithm to decide on a route. We had heavy discussions regarding this part of the added requirements but, fortunately, we were able to decide upon the best alternative that we think will help our implementation and its stakeholders the most.