Design Studio 2

Subject	**Design Studio 2**	
Class	**Informatics 121: Software Design 1**	
Professor	**André van der Hoek**	
Date	**November 1st, 2019**	
Team	**Session #2, Team 9**	
Members	**Name**	**Name**
	Sazeda Sultana (ID# 18592717)	**Kyla Jacqueline Yee (ID# 41209577)**
	Kenny Deland Jue (ID# 61581615)	**Eduardo Magdaleno (ID# 64296561)**
	Ryan Gragasin Cox (ID# 31953949)	

Table of Contents

Customer Prompt

You are tasked with creating an educational traffic flow simulation program. Your client for this project is Professor E, who teaches civil engineering at the University of California, Irvine. One of the courses she teaches has a section on traffic signal timing, and according to her, this is a particularly challenging subject for her students. In short, traffic signal timing involves determining the amount of time that each of an intersection's traffic lights spends 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's traffic lights. This can be a very subtle matter: changing the timing at a single intersection by a couple of seconds can have far-reaching effects on the traffic in the surrounding areas.

Audience

Students: students will be the main audience for this simulation program because they are the target group of people that Professor E is intending the program for.
Professor: The professor will be an audience member as well because they are directly involved in the process of the development, including creating criteria for which the program revolves around in order for their students to be able to learn.

Other Stakeholders

UC Irvine: The school is a stakeholder because they may need to fund the development of the traffic simulation program and may have an influence on the budget given to create it.

Goals

- ***To create an educational traffic flow simulation program***
  - ➢ This traffic flow simulation program aims at being educational and informative so that it becomes capable of facilitating learning about traffic flow through the simulation process. To achieve this, students will be able to gather data from the simulation on traffic depending on what they changed.

- ***To simplify the complexity of traffic signal timing concept***
  - ➢ This program aims at simplifying the idea of traffic signal timing, such as how to regulate the amount of time each traffic lights employ being different colors, for example, green, yellow, and red, to allow cars to flow through the intersection from different directions in a fluid means and without any complexity.

- ***To convert abstract traffic signal timing idea to a concrete one***
  - ➢ As the concept of traffic timing can be abstract, it is crucial to identify, understand and communicate this abstract concept carefully, focusing on proper steps. Through such steps, this program intends to identify and convert this abstract concept into a concrete concept so that it is understandable to everyone.

- ***To Provide new software to play with different traffic signal timing schemes***
  - ➢ This program will be new software for people to play with various traffic timing schemes thus assist with having practical experience on how a traffic flow in real-world scenario works.

- ***To create a visualized map of a map of a traffic area:***
  - ➢ This program must be capable of creating a visual map of a traffic area that students can use by laying out roads in a pattern of their choosing.

- ***To create a non-complex program:***
  - ➢ As the intention of the program is creating a non-complex and simplified program for students so that professor E's students can understand the concept of traffic signal timing straightforwardly.

- ***To generate adjustable features in the simulation program specifically for students:***
  - ➢ Professor E's students must be able to operate and adjust different behaviors of the intersections though through which specific features students will do, such can be fashioned in any way that helps achieve this goal.

- ***To illustrate the influence of traffic signal timing concept:***
  - ➢ This program must be influential enough for students so that students get motivated to use the program whenever they wanted to explore and get clarified on the impact of traffic signal timing on traffic.

Constraints

- A combination of traffic light states that would result in car crashes are not allowed.

- Intersections can only be 4-way intersections, and other intersections ("T" intersections, one-way roads), are not allowed.

- Students can only run one simulation can be run at a certain point in time.

- Cars can not move at different speeds, thus restricting cars from passing each other. Each car must move at the same speed.

- Other traffic conditions, including by not limited to: pedestrians, bridges, accidents, or road closures, can not be added to the simulation, keep it focused on cars and traffic lights.

- Roads can only be horizontal or vertical.

- Roads can only be bidirectional and 2-laned.

Assumptions

- It is assumed that the educational traffic flow simulation program will produce a visual map of any area, laying out roads in a pattern according to students' choice.

- The produced visual map will be simple to understand so that students can use the program with minimal effort; nevertheless, it is expected that this simple and easy to follow graphic map will work substantially to allow for roads of fluctuating length to be placed and generate intersections with different types of arrangements.

- To prevent allowing any consequence of car crashes on an intersection, the program will never have any combination of traffic light states.

- Every intersection on the visual map will have only any 4-way traffic, and there will be not "T" intersections or one-way roads at all.

- Every single intersection will have traffic lights without any stop signs, overpasses, or any other variations.

- Having any sensor on the program will not leave any space to have any effect on students' design of each interaction. Regardless of having any sensor, students' design will be able to detect whether any cars are present in a specific lane. Students will have the freedom to decide if they prefer sensors in the interactions or not.

- Students will be able to simulate traffic flows on the map via the created map mechanisms and the interaction timing schemes.

- Users will experience real-time visualization of traffic levels.

- The program will demonstrate the current state of the interactions' traffic lights. After every alteration of the lights, the state will be updated.

- Changing the incoming traffic flow will be through controlling the number of cars entering the simulation at each of the roads on edge. Students will be able to choose how they want the change to be.

- There will be only one simulation running at a time.

- There will be no variation in different car's speed; it will always be constant so that it becomes difficult for cars to pass each other.

- To avoid additional complications or any stumbling blocks, some features will be shunned in the program, such as pedestrians crossing the road or bridges, any accidents, road closures, etc.

- The programming language used for the implementation must be object-oriented and support at least single level inheritance.

- The user cannot remove Intersections that are adjacent to 2 or more intersections, since the behavior to “bridge the gap” between any adjacent Intersections is undefined.

- Only horizontal or vertical roads will be formed in the program. Any diagonal or winding roads will be circumvented.

- The simulation will not handle any left-turn traffic lights, only regular ones.

Ideas

As a group, we formulated the ideas for our design approaches of the project on keeping in mind and focusing on what experts do while working on a software design that we learned in this class.

- ***Focus on the essence***
    - While working on our approaches, we focused on the essence or of the problem. Every single step we took towards our approaches focused on core sets of considerations, such as how effectively an educational traffic simulator can be created. To accomplish this, we came up with ideas trying to figure out how students can learn from this program.
    - Ideas: Average wait time, Colorful GUI that represents how traffic is, keep track of how long a car takes to cross, number of cars waiting at an intersection.

- ***Prefer solutions that they know work***
    - While generating alternatives, we investigated what solutions have already been made to develop a simulator program like ours. We adopted ideas for the solutions and moved on to other parts of the design project. We looked at multiple simulator to see how they ran, most of them included a lot of option to change specific things about traffic. Here is where we got with ours.
    - Ideas: Navigation apps like Google Maps use colors signify how the traffic is, we want to do that as well, have a simulation actually run and be able to see what the result is. Also to keep track we should use built modules that keep track of time instead of creating one ourselves.

- ***Address knowledge deficiencies***
    - To reach our specific goals of this design project, we filled in with information that we did not know. As a form of knowledge deficiency, assumptions that we made for our project, we made every effort to verify that our assumptions hold in reality through following our design prompt properly and meeting the goals of the design project. We all had to look up how traffic simulations work and fill in other knowledge deficiencies.
    - Ideas: Those who took Data Structures and Algorithms and understood them were put in a group to help others understand how an approach will be implemented. People who are good at making UML diagrams then were able to discuss with them to see how it can be implemented and seen.

- ***Generate alternatives***
    - To reach the project's expected destination, generating alternatives was one of the crucial steps that our team has taken. By searching these alternatives, we maintained flexibility in the design solution. We explicitly sought, advanced, and evaluated our alternatives throughout every level of the design process.
    - Ideas: One of the first ideas we had was to have a grid-like system using 2-D arrays to keep track of it. From this we wanted to see what data can be gathered. After this approach we wanted to see how we can visualize the traffic using this information. Finally, we found a better solution using nodes, edges to better represent the system and be more efficient to use.

- ***Are skeptical***
    - We, as a team was skeptical to explore the necessary breadth and depth of solutions for our design problems. We were skeptical that our design process leads to the solution of our design problems and meets its objective of providing a new software through which students of Professor E's students will be able to play with different traffic signal timing schemes to explore different scenarios.
    - Ideas: After each idea we questioned ourselves and tried to focus on the essence. We also wanted to make sure that any ideas we had was efficient for the program. In this case we realized 2-D arrays may be inefficient and had to create another method.

- ***Solve simpler problems first***
    - While generating our approaches, we focused on the issues that are simple and easy to solve. We did not think or work on all the approach's steps at once. This idea helped us identify and solve complex problems straightforwardly.

    - Ideas: Classes to be used, roads, intersections, cars, traffic lights, signals. Tick to measure the time and speed cars go. Gather data such as average wait time, number of cars waiting.

- ***Draw the problem as much as they draw the solution***
    - As a team, we recognized that understanding of our design problem and understanding of its solution inexorably deepen as we move forward in the design process. Therefore, we drew the challenges of our design approaches as much as we drew the solutions of them, by focusing not only on that both stay in sync but also, we explicitly used advances in the understanding of one to drive advances in the understanding of the other.
    - Ideas: How can we implement this? What are they asking of us? These were the type of questions we asked ourselves from both sides.

- ***Move among levels of abstraction***
    - We collected ideas through designing solutions at multiple levels of abstraction, concomitantly. Instead of focusing only on the high-level model of our simulation program, we worked on low-level models of the program. This design choice helped us directing the path to move up and down among levels of abstraction. As a result, we were able to use resonances among different abstraction levels to trigger insights and identify problems in our design process successfully.
    - Ideas: First level we thought about is how a traffic lights at intersections work and what makes it a good traffic signal that does not cause traffic to build up. Then we started thinking about how we can create the software for it and write it in code.

- ***Go as deep as needed***
    - In our alternatives, we worked checked thoroughly if our pseudocode and UML diagrams are understandable and demonstrate whether each of the approaches is going to meet the requirements and guarantee to encounter the goal of design project.
    - Ideas: We included all UML diagrams with specific functions and attributes they have, we did not want to go as far as to actual code little pieces of code, but create pseudocode for the most essential functions.

- ***Simulate continually***
    - By looking at the needs of the design project, we imagined how our proposed design would work simulating how different parts of the design plan would

support a variety of scenarios of how students of Professor E's students would play with various traffic signal timing schemes efficiently.
    - Ideas: Coming up with ideas we imagined what a traffic simulator would do. From here we get all the ideas for the basic functions and classes and we were able to build on from that.

- ***Are alert to evidence that challenges their theory***
    - Our team was alert to anything that might challenge and obstruct to meet the requirements of the design prompt. We are always open to information that could indicate that our design might be wrong and might not meet the requirements of the project. We inspected every little issue that might indicate a much larger problem hiding beneath the surface.
    - Ideas: Whenever we came to disagreements or stuff that may cause a problem in the future we sought alternatives. This is how approach 3 was formed when we realized having multiple arrays may cause memory error.

- ***Think about what they are not designing***
    - Our team not only focused on the requirements of the document to accomplish the goals, but we also thought about what our design is not intended to do. In speaking and considering boundaries, we wanted to discover where were we over and under-designing.
    - Ideas: We were not just designing a system to see traffic. People had ideas of left turn, cars passing, pedestrians, but we considered that over designing. We had to focus on being able to see traffic and gather data from changing traffic lights.

- ***Invest now to save time later***
    - We anticipated what issues might emerge later in our design process. To foreshadow alternatives futures and perform a cost-benefit analysis, we determined whether investing now in our design alternatives could save time later.
    - Ideas: We invested as must time as we can to the beginning seeing what we can come up with. We kept an open mind to other ideas to create alternative in other approaches.

Alternative Designs

Approach 1

Approach 2 used a grid system in order to store the data. Each 2x1 piece will consist of an ID, a type, the ID's of the grids next to them, a boolean representing whether it is a starting point, and whether or not it currently has a vehicle inside of it. The ID will be a unique identifier for that individual 2x1 grid. The type will represent whether it is a road, intersection, or nothing. The adjacent grids will be a list containing the ID's of the adjacent pieces. And the vehicle will be a boolean, true if there is a vehicle in that space and false if there is not.

The program will be booted up and start by showing a grid with tools to draw roads, and arrows to show traffic flow. The system will be intelligent enough to change grid pieces to an intersection when roads intersect. If a road does not extend to the end of the screen, it will be extended before the program is able to run.

Overall, the data will be stored in a 2-dimensional array. Each piece of data within the array will represent a 2x1 piece within the system. From this, the system will be able to read the data and determine which piece is placed where and what it's current status is. The array will need to be mutable in order to function because each unit run by will update the array. In order to accommodate 2 lane roads, the system will always store a second lane in the system in the grid piece right below (if drawn horizontally) or to the right (if drawn vertically). This will simulate a road with lanes heading in a direction, and allow the flexibility to have a car in one lane but not the other.

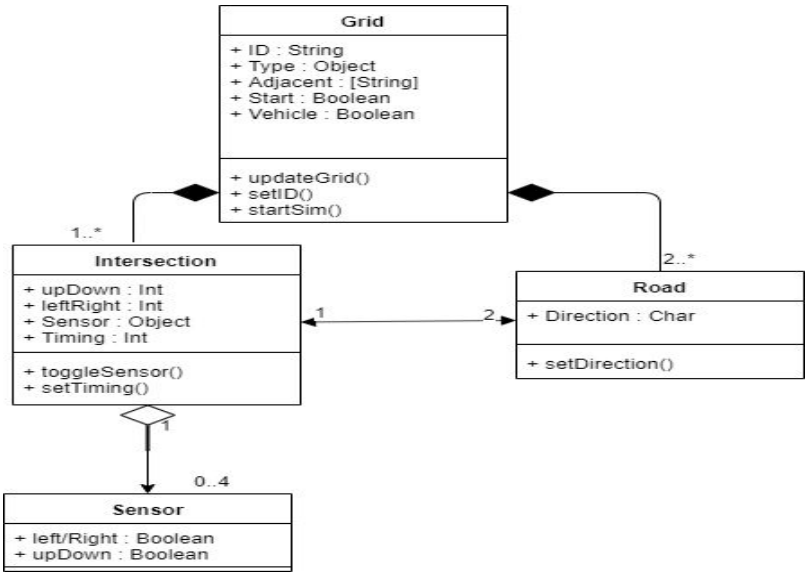
Constraints:

- Each road will need to go from one edge of the screen to the other
- Cars move at the same speed
- Grid-like system will constrain the user to only use two-lane roads
- The tick system will not represent a real unit of time as ticks can be changed to pass at whatever unit of time is desired.

Assumptions:

- The UI will be designed as a grid-like system in order to accommodate this style of storing data
- This will be stored as a 2 dimensional, mutable array
- Turning will not be allowed
- Drawing a road will result in a two-lane road going one direction
- Users do not need to have a bidirectional road

- Cars will not change lanes
- Cars can not turn at intersections. They are only able to drive straight from one end of the screen to the other.



- Traffic
  - Will be able to queue traffic from the start of a road
  - Ability to control how often a car comes in, each lane will have its own frequency
    - 1 every 3 grids, 1 every other grid, etc.
  - Will also have the ability to randomize between a range of numbers how often car comes in
- Timing
  - Ability to control how many grid spots a car will move per second
  - Can pause the simulation and step one grid at a time
- Updates
  - Every time a tick is executed, cars will move in the direction specified and update the mutable array

Approach 2

Approach 2 uses a similar data structure from approach 1 and uses the information gathered from the current traffic simulation to allow students to gather and interpret the data in a different way. This approach gathers the statistics from the current simulation and provides a way to visually see what is going on in the GUI. For example, if a road becomes congested with cars, then the cars will reflect the color of the traffic. If the traffic is light, cars will be green, if the

traffic is building up, cars turn yellow, and if traffic becomes heavy due to an influx of added cars or extended red lights, then cars will turn red.

How this would work is that each road will have statistics in a class. The Road class keeps track of how many cars are currently on the road and the average amount of ticks it takes to cross the intersection at the light. Depending on that time, there will be a constant *threshold value* in which the cars on that road will begin to change color. Roads will have an attribute that will calculate this information. The threshold value set (by the developers, not the student) can be compared with the number of ticks it takes to cross the intersection for cars, or the number of cars on a specific road at a certain point in time that want to cross the intersection.

From this information, the GUI will continuously check the wait times. If the wait time or number of cars passes a certain threshold, the GUI will reflect that by changing the colors of the cars that are on that road. The car class will also have another attribute called Color. This color attribute holds a string that will serve to reflect this type of data *("R" for red, "Y" for yellow, and "G" for green).*

Other than being able to visually see what is happening, another way to gather data is the ability to be able to track a car. If a user wants to know exactly how long it takes for a new car to go across the entire simulation, it has the ability to highlight the car they want to track before they place it into the simulation. There will be a flag that shows the tracked car as it moves through the roads, and the student will be able to see the statistics on the screen on the time it takes for the car to go through the roads both during the simulation.

Data that can be Gathered:

1. The average time it takes to cross an intersection
2. How many Cars are waiting to cross an intersection
3. Number of cars wanting to cross
4. Difference between signifier and no signifier at an intersection.

Constraints:

(in addition to the constraints that apply to all three approaches)

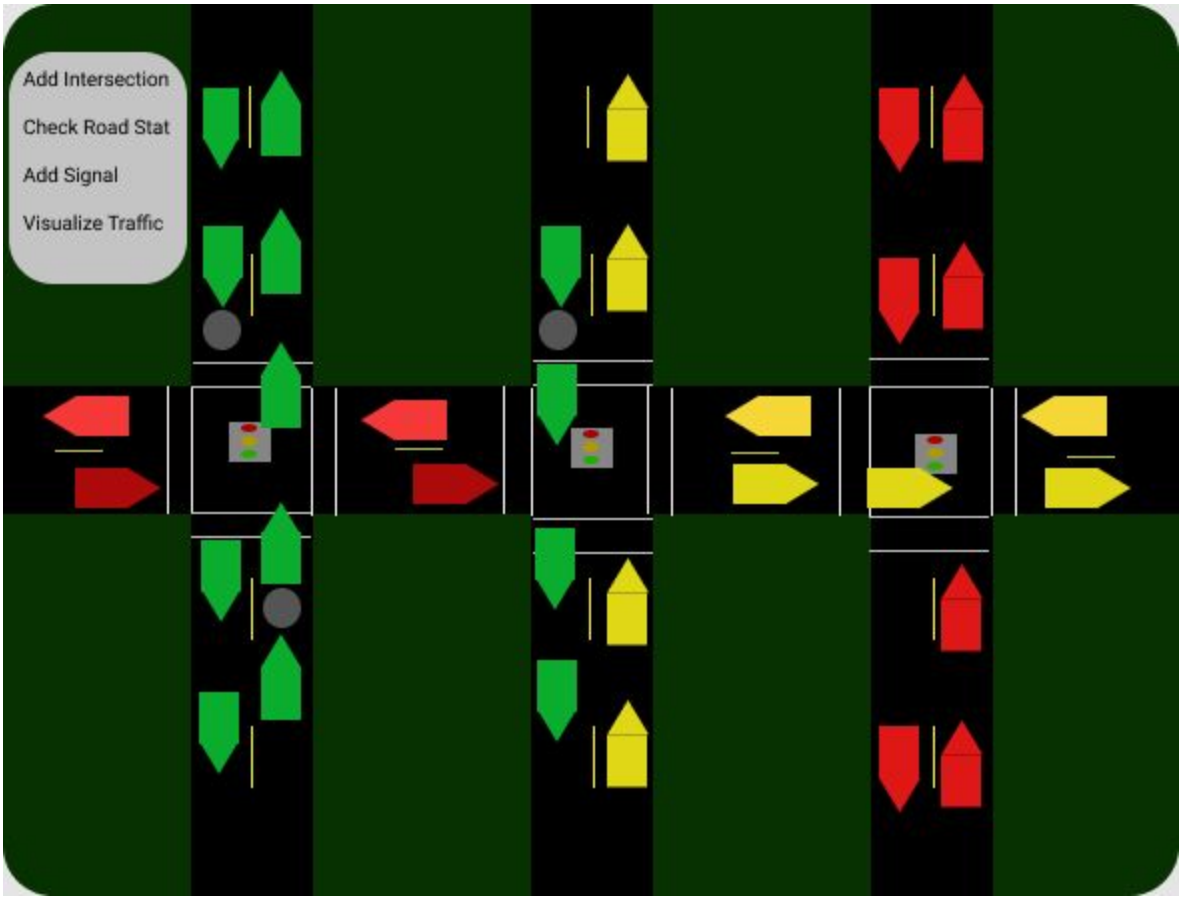
- Students can only choose to track a certain car at the beginning when they are inserting a car into the simulation.
- Students can not change the threshold value in which cars will begin to change colors.

Assumptions:

(in addition to the assumptions that apply to all three approaches)

- Students do not have the functionality to remove the tracking flag from the car until it has successfully exited the simulation.
- There is a constant threshold that determines the color of the car that students are not able to change.
- Students can turn on the feature to see colors of cars at any point during the simulation.

Mock-up:



From the mockup, the user is able to visualize how controlling the traffic light may affect certain intersections. This approach still allows to gather actual data, such as average wait time for a car to cross an intersection, the number of cars waiting in a road, and other data that the student user needs to see how what they are doing either creates or removes heavy traffic. In this scenario from the mock-up, the roads that have censors have a good flow of traffic, because of the signal on the road. Intersections that do not have signals, but have a good flow in traffic lights timing do not create heavy traffic but have a yellow color to signify light traffic. Intersection with nothing to improve traffic have red cars to signify the heavy traffic.

Pseudocode:

```
public void carColor()
{
        if avgWaitTime >= red_threshold:
                change color to red
        else if avgWaitTime >= yellow_threshold:
```

```
                change color to yellow
        else
                change color to green
}

public double avgWaitTime()
{
        int sum
        for all cars on the road
                sum+=car.waittime
        return sum/road.length
}
```

Approach 3

(Chosen Approach)

Architecture and Implementation

This approach uses a graph data structure to implement the relationships between all intersections and roads. This graph could later be easily translated into a visual map of the area, like in a GUI. All “Nodes” serve as “Intersections” and “Entry Points” on this graph and “Edges” represent two-laned “Roads” that “Cars” will be able to travel through.

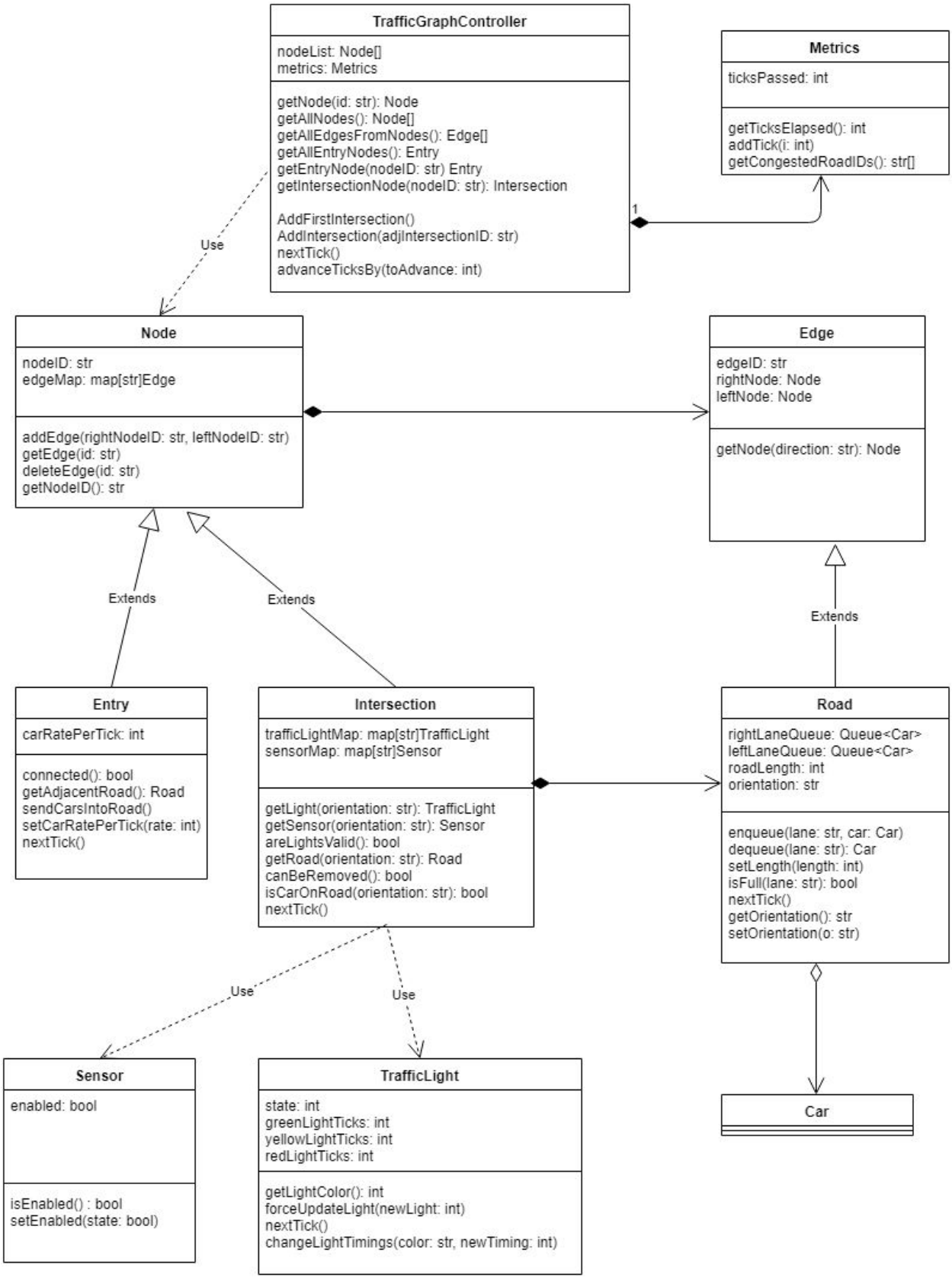

TrafficGraphController
nodeList: Node[]
metrics: Metrics
getNode(id: str): Node
getAllNodes(): Node[]
getAllEdgesFromNodes(): Edge[]
getAllEntryNodes(): Entry
getEntryNode(nodeID: str) Entry
getIntersectionNode(nodeID: str): Intersection
AddFirstIntersection()
AddIntersection(adjIntersectionID: str)
nextTick()
advanceTicksBy(toAdvance: int)
Metrics
ticksPassed: int
getTicksElapsed(): int
addTick(i: int)
getCongestedRoadIDs(): str[]
1
Use
Node
nodeID: str
edgeMap: map[str]Edge
addEdge(rightNodeID: str, leftNodeID: str)
getEdge(id: str)
deleteEdge(id: str)
getNodeID(): str
Edge
edgeID: str
rightNode: Node
leftNode: Node
getNode(direction: str): Node
Extends
Extends
Extends
Entry
carRatePerTick: int
connected(): bool
getAdjacentRoad(): Road
sendCarsIntoRoad()
setCarRatePerTick(rate: int)
nextTick()
Intersection
trafficLightMap: map[str]TrafficLight
sensorMap: map[str]Sensor
getLight(orientation: str): TrafficLight
getSensor(orientation: str): Sensor
areLightsValid(): bool
getRoad(orientation: str): Road
canBeRemoved(): bool
isCarOnRoad(orientation: str): bool
nextTick()
Road
rightLaneQueue: Queue<Car>
leftLaneQueue: Queue<Car>
roadLength: int
orientation: str
enqueue(lane: str, car: Car)
dequeue(lane: str): Car
setLength(length: int)
isFull(lane: str): bool
nextTick()
getOrientation(): str
setOrientation(o: str)
Use
Use
Sensor
enabled: bool
isEnabled() : bool
setEnabled(state: bool)
TrafficLight
state: int
greenLightTicks: int
yellowLightTicks: int
redLightTicks: int
getLightColor(): int
forceUpdateLight(newLight: int)
nextTick()
changeLightTimings(color: str, newTiming: int)
Car

Figure A: UML diagram of the traffic simulation architecture described in this approach

The Tick Algorithm

This traffic simulation relies on a “tick” algorithm that represents an arbitrary time unit. Every time a tick advances, the state of the traffic across all the `Intersections` and `Roads` will update. While ticks timings may differ for the duration of a traffic light state and incoming traffic produced at the `Entry` Points, `Car` speed remains constant and at most one `Car` from each lane in a `Road` will always attempt to cross an `Intersection` per tick.

The following demonstrates how the tick algorithm works:

1. From the topmost level of the program, `TrafficGraphController`, call `advanceTicksBy` or `nextTick`. If `advanceTicksBy` is called with a number greater than 1, repeat all following steps except this step that many times. These two mentioned methods will thus propagate to every other object stored in the graph that has a `nextTick` method.
2. Access the `nodeList` and find all `Node`s (which can be `Entry` or `Intersections`) and `Edges` (which are always `Road`s) using a graph traversal algorithm such as depth-first search or breadth-first search.
3. From all `Node`s and `Edge`s call the `nextTick` function.
    a. If the `Node` is an `Intersection`, `nextTick` will call all four `TrafficLights`’ respective `nextTick` to ensure the lights change state depending on user configuration of the tick timings.
    b. If the `Node` is an `Entry`, it will produce some arbitrary amount of `Car`s into an adjacent `Road` depending on the user configuration.
    c. If it is an `Edge` (meaning it will be a `Road`), the `Road` will move the car to the proper road depending on the `Intersection` state.
    d. More details about the `nextTick` implementations can be found under the *Entities* section.
4. Call `addTick` in the `Metrics` instance with the corresponding value that `advanceTicksBy` was called with, or with a value of 1 if `nextTick` was called.
    a. This is for keeping track of the simulation metadata in case the user ever wants to access it.

Entities

Node

`Node` objects contain some rudimentary attributes:

A. `nodeID`, which is a unique identifying string for that node.

B. `edgeMap`, a simple dictionary-like data structure that contains `edgeID`s that maps to their respective `Edge` instances. These `Edge` instances link to adjacent `Node`s on the graph.

By accessing these `Edge`s, it is possible to find adjacent `Node`s connected to the opposite end of the `Edge`. `Node` contains some methods to get, add, and delete `Edge`s. `Node′s` methods and attributes are inherited by `Entry` and `Intersection` classes.

Edge

The `Edge` contains a unique `edgeID` string for the purpose of identifying the specific edge from a `Node`. It also contains a `rightNode` and `leftNode` attribute, both of which represent adjacent `Node`s to the `Edge`. An `Edge` serves as the link between two `Node`s in the graph.

Road

`Road`s inherit behavior from the `Edge` class. The `Road` acts like an `Edge` but also contains additional data structures:

A. `rightLaneQueue` and `leftLaneQueue`. These queues represent the 2 opposite lanes of the road and enqueue and dequeue `Car`s in the first-in-first-out (FIFO) order. When a `Car` is enqueued, it "enters" the lane and when it is dequeued, it "leaves" the lane.
B. `roadLength`, which defines the maximum length each queue can grow to. If a lane becomes full and another `Car` attempts to be enqueued into this lane, the `Car` entry is denied.
C. `orientation`, which can be "horizontal" or "vertical". This attribute is especially important because it provides information on which `TrafficLight` would be facing each lane in adjacent `Intersection`s.

Every `Road` implements a `nextTick` method which will be called for every new tick the traffic simulation advances. As mentioned in the tick algorithm above, at most one `Car` from each lane can leave the `Road` every tick. A more detailed behavior of a single `nextTick` call is specified as follows:

1. On a `Road`, access an adjacent `Node` (`leftNode`, `rightNode`) inherited from `Edge`.
2. The `leftNode` will be the traffic direction of the `leftLaneQueue`, and the `rightNode` the direction of `rightLaneQueue`.
3. If the adjacent `Node` is an `Intersection`, check the proper `TrafficLight` which will be calculated with the `orientation` attribute. (For example, if the `orientation` is "horizontal" and the `rightNode` is an `Intersection`, then the `TrafficLight` state of the "west" will be retrieved. More info on this in the following subsection.)
    a. If the `TrafficLight` is green or yellow, dequeue the oldest `Car` and enqueue it to the corresponding `Road` lane corresponding to the *compass* orientation of the `TrafficLight`. (In other words, if the `TrafficLight` is oriented to the "west" then the car will enter the "west" facing road.)
4. If the adjacent `Node` is an `Entry`, dequeue the oldest `Car` from the proper lane and drop it from the simulation.

Intersection

`Intersection`s are a class that inherits properties from the `Node`. An `Intersection` abides by several properties:

A. It is connected to four and only four `Road`s, which are extended from `Edge`s. These `Road`s have orientations of “north”, “south”, “east”, and “west”. They are stored on the `edgeMap` inherited from the `Node` class.
    a. It is most accurate with respect to this implementation of the traffic simulation to envision an `Intersection` as a plus-shaped, four-way intersection. (Refer to **Figure B** below for the visualization)
B. The opposite side of each `Road` must be linked to an `Intersection`, or if not, then an `Entry Node`.
C. `Intersection` will never manage `Car`s, just the timings of its four `TrafficLight`s.
D. The `sensorMap` contains the same orientations of “north”, “south”, “east”, and “west”. Each `orientation` maps to a `Sensor` object, whose purpose is to provide info every tick to a corresponding `TrafficLight` of whether there are cars on the road. `Sensor`s can be enabled or disabled for every orientation.
E. The `trafficLightMap` has the same four orientations as `sensorMap` but instead maps to a TrafficLight object.
F. An `Intersection` cannot be removed from the simulation if it is adjacent to two or more `Intersection`s. If an `Intersection` is removed from the graph successfully, it is substituted with an `Entry Node`. To remove an `Intersection`, access all its connected `Edge`s and replace the connection with an `Entry Node`. The `Intersection` will thus be isolated and inaccessible forever unless linked again.

It is important to note that a “north” oriented `TrafficLight` would be facing the “south” `Road`, while the “north” oriented `Sensor` would be on the “north” `Road` and provide sensory data to the “south” `TrafficLight`. All four `TrafficLights` can have their state updated to change the timing of lights based on a user-specified tick input. However, if the timing conflicts between the lights to make a car accident possible, then the method `areLightsValid` will return false, warning the user that they have an invalid timing configuration.

When a car wants to travel through an intersection, it inspects the light, and if it is yellow or green, then it can pass through to the opposite road using the `getRoad` method. Also, if the sensor of the road is enabled, then the `Intersection` will internally convey that data to corresponding `TrafficLight` and force the state to update.

`Intersection`s contain internal logic for handling ticks, located in the `nextTick` method. This method must perform the following actions at minimum:

1. Inspect every `Sensor` in the `Intersection`.
2. For the enabled `Sensor` orientations, retrieve the `Road Edge` corresponding to that orientation.

3. Inspect the *relative* right lane of the `Road` with respect to the `Intersection`. If there is one or more `Car`s on the lane, access the opposite oriented `TrafficLight` and change its state correspondingly, overriding default light timings.
    a. By “relative”, imagine a point in the center of a plus-shaped intersection. With respect to the center, sometimes the *right* lane of a `Road` connected to the intersection will actually be the *left* lane of a `Road` from the `Road`’s perspective. Refer to the following **Figure B**:

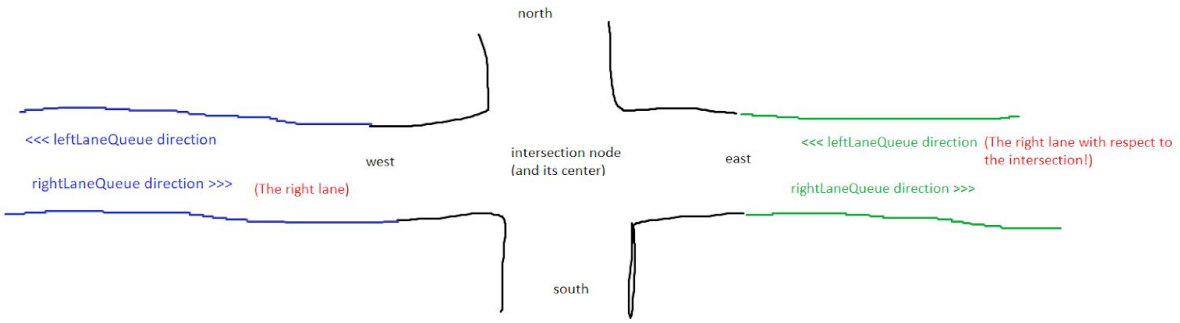


Figure B: The blue and green drawn areas are two different `Road`s. From the green `Road`, observe how the `leftLaneQueue` would be acting as the right lane for that `Road` leading into an `Intersection`. The blue `rightLaneQueue`, however, acts as the right lane as expected.

`Intersection Node`s do not ever actually process `Car`s directly - it just handles `TrafficLight` state. This responsibility is delegated to `Road`s, and will be demonstrated later in the document.

Entry

The other class that inherits from `Node` is `Entry`. `Entry Node`s serve as nodes that exist on the border of the simulation map - the graph created by `Intersection`s and `Road Edge`s. They have some distinct properties:

A. `Entry Node`s can produce 0 or more `Car`s every tick in the simulation using the `sendCarsIntoRoad` method.
B. `Entry Node`s can change its `Car` production rate using `setCarRatePerTick`.
C. Each individual `Entry` is connected to exactly one `Road Edge`. This `Road` is where the `Car`s will enter.
D. Any incoming `Car`s to an `Entry` will be consumed and exit the simulation.
E. `Entry Node`s are never explicitly removed and added by a user; when an `Intersection Node` is placed, the entry `Node` will be added automatically. `Entry Node`s also mark possible positions where another `Intersection` can be placed. A placed `Intersection` will replace the `Entry Node`, and when the `Intersection` is removed, the reverse occurs.

Like `Intersections`, `Entry Node`s also contain their own `nextTick` method. When an `Entry`'s `nextTick` is called, the following happens:

1. Retrieve the only connected `Road` to the `Entry`.
2. Depending on the configured `Car` production rate per tick, create that many new `Car` objects.
3. For every new `Car`, attempt to send it in to the corresponding right lane of the `Road` by enqueuing them.
4. If the corresponding right lane's queue is full, it does not enqueue.

Car

`Car`s are the data that move between the `Node`s and `Edge`s. A `Car` has no special properties or attributes; it is a simple object instance. A `Car`'s behavior is never directly modified by the user. How it acts will be affected by the user's `TrafficLight` and `Entry` timings across the simulation.

TrafficGraphController and Metrics

The program is initialized through the `TrafficGraphController`. Only one instance of this class will live throughout a single traffic simulation. It is also home to a `Metrics` instance, which is used to retrieve statistics for that program execution duration. `Metrics` provides traffic simulation data including but not limited to the total elapsed simulation ticks and the `Road`s that are currently congested. `TrafficGraphController` exposes the logic to create new `Intersection`s. Like the many other entities that exist in this implementation, it also contains its own `nextTick` function. `nextTick` behaves as follows:

1. Call its `getAllNodes` method.
2. For every `Node` returned in the `Node` array, if the `Node` implements a `nextTick` method (such as `Entry` or `Intersection`), call it.
    a. Ideally the `Node` with `nextTick` would contain logic to propagate `nextTick` calls to all its internal data structures if applicable.
3. Call `getAllEdgesFromNodes`, which traverses all the `Node`s in the graph to retrieve the connected `Edges`.
4. For every `Edge` in the traffic simulation returned from the said method, if the `Edge` implements a `nextTick` method, call it.
5. Update the `Metrics` instance with the incremented tick.

`advanceTicksBy` can be called in alternative to `nextTick` with a numerical value representing the number of simulation steps to take. In the case that the said method is called, simply repeat the above algorithm that many times.

Example Scenario

With these entities of the architecture explained, a simple scenario of `Car` movement over two ticks would behave like this:

1. A `Car` instance is produced by the `Entry Node` on the first tick. The `Entry Node` will access the Road Edge it is adjacent to and access the queue with the method `enqueue`. Arguments for enqueue would be “right” and the `Car` just produced, in this case.
2. The `Car` is enqueued into the lane successfully, as the lane is empty.
3. The second tick occurs when `nextTick` is called on all relevant objects.
4. Using the `Road`’s inherited `getNode` method from `Edge`, get the `Node` opposite of the `Entry Node` using by calculating the orientation with whether the `Road` is aligned horizontal or vertical. This would be an `Intersection Node`.
5. Assuming the `TrafficLight` opposite of the `Road` is green or yellow, the `Car` will be dequeued from the relative right lane with respect to the `Intersection` and enqueued into the proper opposite `Road` lane that the `Intersection` is connected to.
    a. If the `TrafficLight` is red, the `Car` will not be dequeued from that lane.

In the case where the any lane is full, the `Metrics` instance will notify the user of a congested road, and there will be a no-op on the affected lanes.

Rendering Data

This implementation provides certain entry points to render traffic simulation data onto some graphical user interface. Here are some examples (though not exhaustive).

- A graphical map of all `Intersection`s and `Road`s can be created using the `getAllNodes` and `getAllEdgesFromNodes` methods in `TrafficGraphController`.
    - With these `Node`s, it is possible to individually modify traffic density for every road on `Entry Node`s using `setCarRatePerTick` in the respective `Entry` instance.
    - It is also possible to change individual `TrafficLight` timings and enable and disable sensors from `Intersection Node`s via public methods such as `getLight`, `getSensor`, `setEnabled`, and `changeLightTimings` from `TrafficLight`, `Sensor`, and `Intersection` instances.
- The user can advance the simulation by calling the `TrafficGraphController`’s `nextTick` or `advanceTicksBy` via some button or other user interface component.
- Users can be notified of traffic congestions by calling `getCongestedRoadIDs` from the `Metrics` instance.
- `Cars` on `Road`s can be visualized by checking the length of each lane queue.

We chose Approach 3 to be our final approach. Although the final approach we decided to go with would be more difficult to implement, it would provide a more robust system compared to a 2D array, which Approach 1 and 2 uses.

Trade-offs (Approach 3 - *final approach* **vs**. Other Approaches)

- Approach 3 may require complex memory management if there is no garbage collector to prevent memory leaks in deleted graph node.
- It could be costly to implement graph data structures and traversal algorithms, unlike simply iterating through a grid (in Approach 1), which uses a 2D array.
- Using the 2D array from Approach 1 would cause the simulation to be choppy since pieces of data have to exist within the grid, creating a less smooth experience for the user.
- The final implementation will likely be more costly due to its higher degree of difficulty, meaning a more expensive engineer will be tasked to build it.
- The 2-dimensional approach is a low abstraction level approach that could be assembled quickly.
- The GUI implemented in approach 2 could be used in either approach 1 or 3 as a way to influence design ideas, but could not be used by itself. It was built from approach 1 but can be integrated into approach 3.
- Approach 3 may take a team to build, whereas approach 1 could likely be built by one engineer.
- Approach 3 may require extra team members to maintain the system.
- Approach 3 could collect analytics on the system more efficiently than approach 1, allowing possibilities for expansion into a dashboard system to let users visualize data easier.
- Approach 3 has a higher likelihood of bugs compared to approach 1 or approach 2 because of its more complex nature.