William Yang
Dennis Li
Payam Dowlatyari
Kenzo Spaulding
Tyler Mos

Traffic Simulator Architecture & Implementation Design:

Overview and Essence:

This design document describes the basic structures and algorithms for a traffic signal simulator. We intend this simulator to be an educational tool that gives students the ability to see how much changing even one stoplight can affect everything around it. The following parts of the document will come to describe how we've designed this simulator and why we believe these design choices allow for a great learning experience for students.

Goals (New to this Implementation):

- Replayability (students can rewatch the same thing they just saw over again)
    - Students can also reload a past simulation
- Easily editable: Users should easily be able to add/remove/ move objects
- Scraping relevant information to give to the student in real-time, such as time and traffic density of each road
- Play multiple simulations simultaneously

Constraints:

- Traffic lights must be editable (timings, a sensor that says if a car is in a lane)
- Protected left turns and a special lane for turning left
- No crashing allowed as a result of light timings
- 4-way intersections only, no one-way roads, no T intersections
- Save simulations exactly as they happened to view later
- Have multiple simulations run simultaneously
- Both trucks and cars (different movement speeds)
- Can have multiple lanes so cars can pass trucks
- Can have road closures

Assumptions:

- Computer is capable of managing the load.
- Students are able to install and use the software.
- Students are willing to learn how to interact with the software.

Audience:

- Students: Students will use this program in a lab setting to see how traffic occurs in real-time.
- Professors and TAs: The professor and TA will likely walkthrough students how to operate this program.

- IT Staff: It is likely that tech staff will need to install this software onto computers, as it is unlikely that lab computers will come prepackaged with this software. They will also be responsible for required to update and maintain
- Front-end developer: As we are designing everything but interaction design, front-end developers will be interacting this program, and try to further design to match the needs of the academy.

Stakeholders:

- UCI: The university will be the entity who distributes the program amongst lab computers and those who plan to use it at home.
- Students: Students will use this program in a lab setting to see how traffic occurs in real-time.
- Professors and TAs: The professor and TA will likely walkthrough students how to operate this program.
- IT Staff: It is likely that tech staff will need to install this software onto computers, as it is unlikely that lab computers will come prepackaged with this software. They will also be responsible for required to update and maintain

Ideas:

- Graph structure (directed graph) where cars will use the structure to find the fastest route to their destination node. This will allow for cars to use pre-existing efficient graph algorithms and easily handle road closures.
    - Pros to this structure:
        - Easy to manage road closures,
        - easy to handle left turn logic,
        - easy to manage no randomness,
        - easy to recreate,
        - easy to edit,
        - easy to build
    - Cons to this structure:
        - Difficult to build light sensors
        - Difficult to handle passing
- Nested list containing objects that build together to make a map. Each grid space either contains nothing or an object that cars/trucks can be contained on (like a road segment)
    - Pros:
        - Easy drag and drop grid
        - Easy to make single lane roads
        - Easy to make sensors
    - Cons:
        - Difficult to make multi-lane roads
        - Difficult to left turn
        - Difficult to route plan

- MVC architecture for the front-end to control building up the model and displaying it to the user.
    - Model has two parts, one is the current model being built and one is a database of all previous models saved. The MVC can be duplicated into multiple windows and all have access to the same database so models can be played simultaneously side-by-side.
- Database of past models (RDBMS vs relational)
    - Pros of RDBMS
        - Structured, easy to manage and handle fast
    - Cons of RDBMS
        - Regimented, strict
    - Pros of Relational
        - Easy to build up, can handle lots of random types of data
    - Cons of Relational
        - Slower, a bit more difficult to find specific data and load it with custom types (since it's stored as json).
- Have a few different data objects that store data and handle interactions.
    - Ideas for types could be roads, intersections, doors (where cars enter/exit scene), cars/trucks (parent class?),

Improvements and Changes Upon Previous Implementation:

Per Professor E's further specifications, our last implementation was not sufficient and needed to be tweaked. The features that had to be added were:

Addition of Left Turns:

Our last implementation only used straight-driving and right-turns. This implementation requires the inclusion of left turns in addition to what we previously had in part 1 of our documentation.

Reproducibility:

The new implementation requires that results can be repeated given the same parameters and data. This is necessary to ensure that a student can change elements of a map and see an immediate impact on how the traffic density changes. This was impossible in our last design, as we made use of random generation and turning.

Multiple Concurrent Simulations - Including Identical Ones:

In our last implementation, we could have easily had multiple simulations ongoing at the same time by producing multiple map objects. However, due to the inclusion of randomness in our last design, it would have been near impossible for two simulations to have exactly the same results, even if they both used the same initial map, car, and traffic light data.

Addition of Trucks and the Ability to Pass Trucks:

This new implementation will require that in addition to the preexisting car objects, trucks object will also be driving along the road. This is not inherently hard to add to our design, as it operates similarly to a car, but this addition affects how cars will now behave. Trucks will move

slower and be heavier than cars by some measure, and cars will need to be able to pass these slower moving vehicles.

Because the above additions required a severe reworking of our previous implementation, the following aspects were added to this one:

Removal of Randomness in Turning, A Change to Strict Algorithms:

The inclusion of random turning was a cheap and easy way to simulate turning in our simulation. However, because cars turned randomly, it was very difficult to reproduce the same results every time. The removal of random turning and the addition of an algorithm that consistently creates the same results with the same information is necessary for a positive learning experience. This also enables cars not to exhibit strange behavior approaching a road closure.

Removal of Random Car Generation:

Going with the fact that randomness in turning made the previous implementation work, but not consistent, it is also necessary to remove the randomness in car generation. If a car spawns randomly, it can be reproduced by only creating cars at the same time as previously simulated, but it would be difficult to simulate two exact simulations on original data.

Destination Determination:

Random turning helped decide where our cars go; they would randomly turn into a destination on their own. With the removal of random turning, our newer implementation requires that the cars' destination be determined before it begins traveling a specific route. This can be done by having a list of origins and destinations shared between all simulations meant to use the same data. This enables not only the reproducing of a simulation, but also allows two simulations to run the same map and car data at the same time and produce the same results.

Snapshots and Event Playback:

For educational purposes, a student should be able to rewatch past actions that have occurred. The map will be scanned per update interval so that the data can be shown

Additional Lanes, Including a Passing Lane (# of lanes chosen by user)

In the original implementation, all roads had two lanes - one for each direction. In the new implementation, since cars need to pass trucks, multiple lanes are now included and will allow cars to pass trucks by switching lanes.

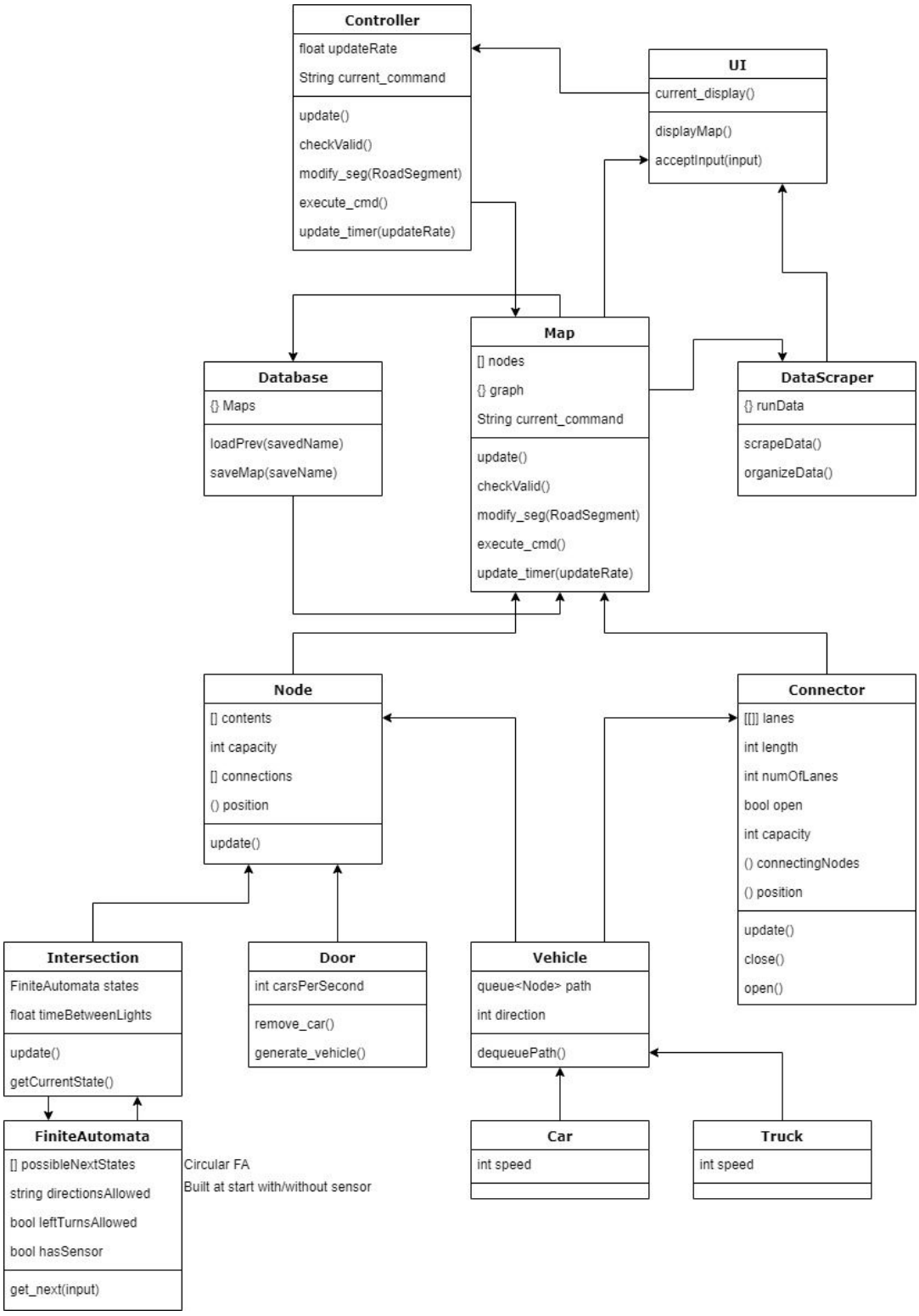
Changes in how a Road is implemented (structure-wise)

Our previous implementation of the road did not permit many of the features listed above, like determining the destination ahead of time or passing through lanes. This then entails a reworking of the code and structure (See below). Roads can be removed to simulate closures, and that is easily done in our implementation.

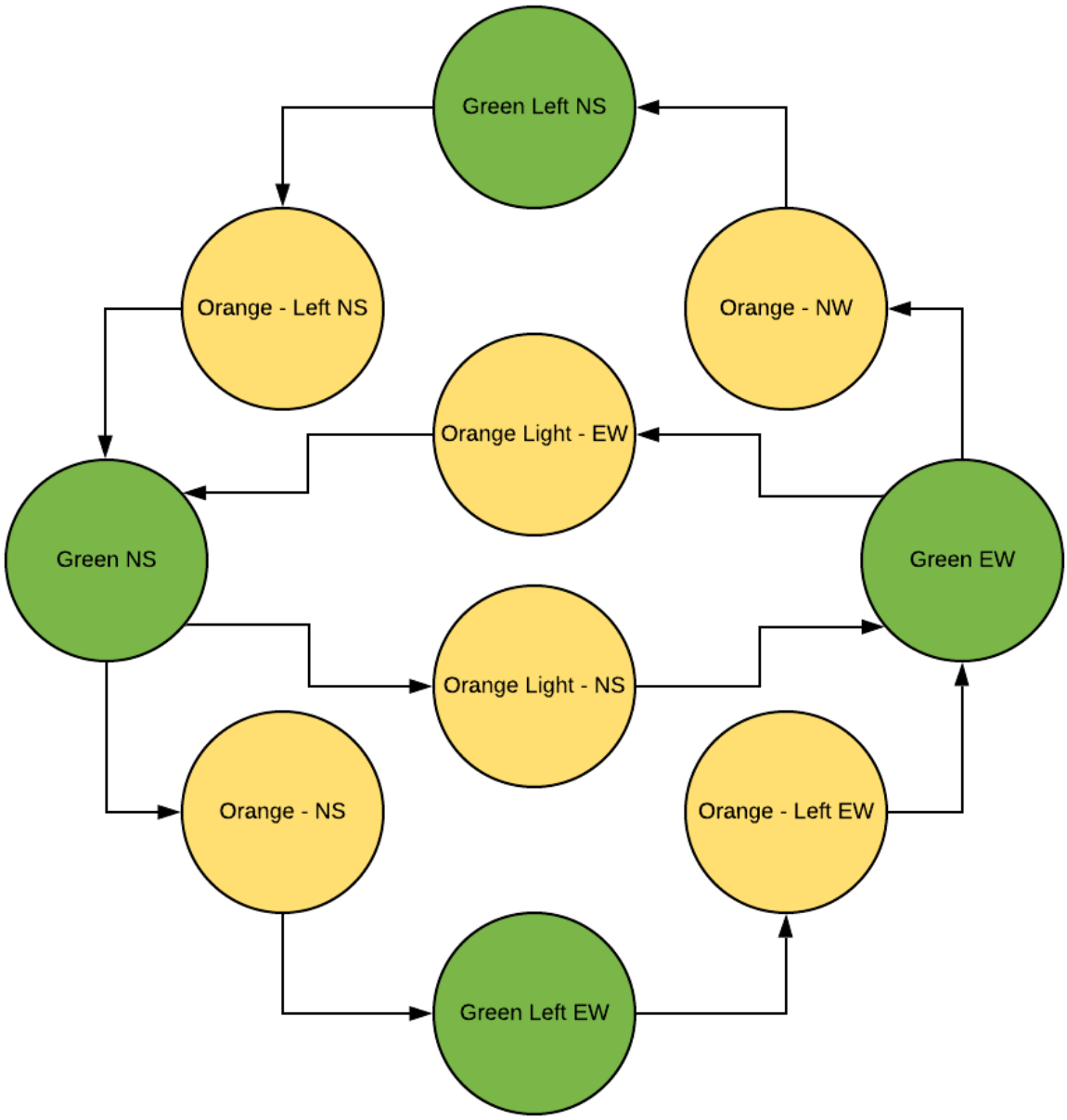
DataScrapper for complete understanding of the simulation

Our previous implementation, students can only learn about the configuration they set up for the simulation through real-time observation. By scraping data and give students an idea of how the traffic performed, they can have a way or method to understand if one traffic setup is better than others.

UML


Controller
float updateRate
String current_command
update()
checkValid()
modify_seg(RoadSegment)
execute_cmd()
update_timer(updateRate)
UI
current_display()
displayMap()
acceptInput(input)
Database
{} Maps
loadPrev(savedName)
saveMap(saveName)
Map
[] nodes
{} graph
String current_command
update()
checkValid()
modify_seg(RoadSegment)
execute_cmd()
update_timer(updateRate)
DataScraper
{} runData
scrapeData()
organizeData()
Node
[] contents
int capacity
[] connections
() position
update()
Connector
[[]] lanes
int length
int numOfLanes
bool open
int capacity
() connectingNodes
() position
update()
close()
open()
Intersection
FiniteAutomata states
float timeBetweenLights
update()
getCurrentState()
Door
int carsPerSecond
remove_car()
generate_vehicle()
Vehicle
queue<Node> path
int direction
dequeuePath()
FiniteAutomata
[] possibleNextStates
string directionsAllowed
bool leftTurnsAllowed
bool hasSensor
get_next(input)
Circular FA
Built at start with/without sensor
Car
int speed
Truck
int speed

Finite Automata Lights :



Description of UML:

Initially, the user builds a map using the controller (MVC) and decides where to put nodes. The nodes are then linked together via connectors to build a graph. **Nodes with one entrance/exit connector are doors which let vehicles enter and exit**. Every car has a destination **door** to which they have a queue of turns to get to that intended door. **Intersections are any node that has**

connectors leaving from more than one direction. This is how we figure out route planning for cars, as there are existing algorithms for the shortest path to a target node on a graph.

There are a few important distinctions to make. Intersections do not have different types to handle left turns and sensors. Instead, the user can edit the intersection's logic, which is a deterministic finite state machine, which can include left turns (which would make the intersection have a left turn lane), or sensors (which would be included in state change parameters).

The next important info is how the logic of connectors (roads) work. This is a vital part of the software because there are multiple restraints having to due with passing, speed, and such. The architecture states for roads to be able to have more than one lane, and for cars to be able to pass trucks. This could be accomplished by having a list of static length Linked Lists where cars go from node to node and can jump from one list to the other. This could also be a static length array (which would make lane change easier) or a linked list with pointers to the lanes next to it. Whatever the implementation is, this is an important architectural decision.

Doors are nodes that have one edge pointed in and one edge pointed out. This is used to insert new cars and trucks into the simulation and remove them. This introduces another **important part** of the simulation. Cars and trucks, when generated, will automatically have a door node chosen to be their destination node - the users can themselves write a list or have the simulation generate it for them. The simulation will then generate a list of directions (which turns they'll take at each intersection) from their current door (of generation) to the destination door (for exit).

The Data Scraper continually analyzes the map, returning important information like traffic density; this information allows the user interface to have easy access to information that would allow it to represent our data structures in a graphical way that users can easily understand and learn from. Additional data it can provide includes calculating the congestion of the traffic by summing how much ticks each cars have stopped, according to its internal counter; a road will be more congested if cars within it have stopped for an extensive time. It will also sum up the number of cars that have exited the simulation; comparing this information with the spawn rates will allow students to identify which of their traffic settings was the most optimal.

Pseudocode:

Vehicle timers are used to measure how long a vehicle has been on a certain stretch of road, should the user be interested in doing so.

Internal Timers are timers stored per object and allows them to perform certain actions.

Algorithm for Determining Origin and Destination:

If user provides a data file of information containing vehicle and map details, use that information.

Else, randomly assign cars origins and destinations and save this information in a file before the simulation begins.

Algorithm for Pathing:

//our pathing algorithm is very similar to dijstra's algorithm

When (car is created){
    If (map has more than one node){
        Have origin node value = 0;
            For (all adjacentNodes){
            Calculate value if the car were to travel to the adjacent nodes;
            Store the value in that corresponding node;
                While(another node exists){
                    Pick next node with minimal distance and repeat node calculations;
                }
                Use (one of) the node path(s) with the shortest minimum distance from the origin to the distance as the path.
    //If multiple paths exist to one end, use the path that makes the most right turns, simply for consistency.
    }
}

Algorithm for Intersection Lights:

    Internal Timer = 0;
    Initial State of the Light = XaxisGreenAndYaxisRed;
    When the Simulation is updated, increment the Internal Timer by 1;
    If Sensors are being used and have been triggered, increment the Internal TImer by 1;

    If (the light is XaxisGreenAndYaxisRed) and (the timer equals time between green and orange){
        State of the Light = XaxisOrangeAndYaxisRed;
        Internal Timer = 0;
    }
    Else if (the light is XaxisOrangeAndYaxisRed) and (the timer equals time between orange and red) and (Left Turns Enabled) {
        State of the Light = XaxisRedAndYaxisLeftTurns;
        Internal Timer = 0;
    }
    Else if (the light is XaxisOrangeAndYaxisRed) and (the timer equals time between orange and red) and (Left Turns Are Not Enabled) {
        State of the Light = XaxisRedAndYaxisGreen;;
        Internal Timer = 0;
    }
    Else if (the light is XaxisRedAndYaxisLeftTurns) and (the timer equals time between left turns and green light){

State of the Light = XaxisRedAndYaxisGreen;
Internal Timer = 0;
}
Else if (the light is XaxisRedAndYaxisGreen) and (the timer equals time between green and orange){
State of the Light = XaxisRedAndYaxisOrange;
Internal Timer = 0;
}
Else if (the light is XaxisRedAndYaxisOrange) and (the timer equals time between orange and red){
State of the Light = XaxisLeftTurnsAndYaxisRed;
Internal Timer = 0;
}
Else if (the light is XaxisLeftTurnsAndYaxisRed) and (the timer equals time between Left Turns and Green){
State of the Light = XaxisGreenAndYaxisRed;
Internal Timer = 0;
}

Algorithm for Doors: (used for vehicles entering/exiting the simulation)

Internal Timer = 0;
When the Simulation is updated, increment the Internal Timer by 1.
Current Lane of the Door to Spawn on = 0; *//inconsequential, but the rightmost lane represents the 0th lane, and the leftmost lane represents the largest numbered lane)*

If (Internal Timer >= time between car spawns){
if(there is only one lane){
If (space ahead is empty){
create cars destination algorithm;
spawn car into the next empty spot and assign algorithm to it;
Internal timer = 0;
}
else if (space ahead is not empty){
Nothing occurs (will wait until the next update to attempt to add a car).
}
}
Else if(there is more than one lane){
If (Internal Timer >= Time between Car spawns){
Lanes Passed = 0;
While (Lanes Passed < Total Amount of Lanes){
If (current lane is empty){
create cars destination algorithm;
Spawn car in current lane;;

```
                                        Internal timer = 0;
                                        break;
                                } else if (current lane  is not empty){
                                        Currentlane++;
                                        Lanes Passed ++;
                                }
                        }
                        If (Lanes Passed = number of lanes){
                                Nothing occurs (wait until the next update to attempt to add a car).
                        }

                }
}
```

Algorithm for Intersections:

// This code is used as a car approaches an intersection.
//In terms of operation order, the intersection pieces with the least amount of cars in front of it acts first. This includes 0 cars, such as right before a door.

```
Internal Timer = 0;
When the Simulation is updated, increment the Internal Timer and the Vehicle Timer by 1.

If (vehicle aims to turn left == true){
        If ((leftTurnLight == green and (( vehicle is a car) ||  (Internal Timer >= time it takes for a truck to move)){
                Have car turn left;
                Internal Timer = 0;
        }
        Else if ((vehicle is a truck ) and (Internal TImer < time it takes for a truck to move)){
                Internal Timer ++;
        }
        else if (leftTurnLight == red ){
                Prevent car from moving forward;
                Internal timer = 0;
        }
}
 else if(vehicle aims to go straight == true){
        If ((straightLight==green) and  (( vehicle is a car) ||  (Internal Timer >= time it takes for a truck to move)){
                Have car proceed forward;
                Internal timer = 0;
        }
        Else if ((vehicle is a truck ) and (Internal TImer < time it takes for a truck to move)){
                Internal Timer ++;
```

```
        }
        Else if (straightLight == red){
                Prevent car from moving forward;
                Internal timer = 0;
        }

)
 else if (( vehicle aims to go right) and  (( vehicle is a car) ||  (Internal Timer >= time it takes for a truck to move)){
        If ((straightLight==green) and  (( vehicle is a car) ||  (Internal Timer >= time it takes for a truck to move)){
                Have car proceed right;
                Internal timer = 0;
        }
        Else if ((vehicle is a truck ) and (Internal TImer < time it takes for a truck to move)){
                Internal Timer ++;
        }
        Else if (straightLight == red){
                Prevent car from moving forward;
                Internal timer = 0;
        }
}
```

Algorithm for Roads:

// In terms of operation order, the road pieces with the least amount of cars in front of it acts first. This includes 0 cars, such as right before a door.

Internal Timer = 0;
When the Simulation is updated, increment the Internal Timer and Vehicle by 1.

```
        If (currentVehicle is in the passing lane){
                If  (space ahead in the right lane is empty){
                        Have currentVehicle return to the right lane;
                }
                Else if (space ahead is empty){
                        Move car to space ahead;
                }
                Else {
                        Do nothing (Attempt to move next update);
                }
        }
        Else If (currentVehicle == car and the space ahead is empty){
                Move Car to space ahead;
        }
```

```
Else if (currentVehicle == car and the space ahead is not empty and the vehicle in the space ahead is a car){
        Do nothing (Attempt to move next update);
}
Else if (currentVehicle== car and the space ahead is not empty and the vehicle in the space ahead is a truck){
        If (there is only one lane){
                Do nothing (Attempt to move next update);
        }
        else if (there is more than one lane and left lane is empty){
                Have currentVehicle move to the space ahead in the left lane and pass;
        }
        else if (there is more than one lane and left lane is not empty){
                Do nothing (Attempt to move next update);
        }

}
else if (currentVehicle == truck  and the space ahead is not empty){
        Do nothing (Attempt to move next update);
        Internal timer = 0:
}
else if ((currentVehicle == truck) and (the space ahead is empty) and (the Internal Timer < time it takes for a truck to move)){
        Internal timer ++;
}
Else {
        Have truck move to the space ahead;
}
```

Reflection:

Why we made certain choices in our design:

Most of the additions featured in this document are additions made because they were requirements. Rather, this part of the reflection focuses on why these aspects were designed this way.

Since the simulation should be reproducible, it is only natural that randomness be removed. It was more difficult to determine what should have replaced it. We settled on determining the origin and destination and creating a path based on that information rather than dynamically creating the path as the car moves. This serves to solve multiple issues we faced at once - we removed randomness, but we also added a means to reproduce data by saving pathing data and a means to run simulations that create the exact same results by utilizing a consistent algorithm.

In order to simulate trucks moving slower than cars, we utilized what seemed like the only option, given our data structure. Trucks move the same amount as cars at a time, but move at different intervals. For example, a car might move once every tick, but a truck may take two ticks to

move. This implementation, when combined with the right graphical interface, will give an illusion of trucks moving slower, even if they are moving after a certain interval of time.

Pros v Cons compared to old implementation:

The current implementation of the traffic simulation software has definite improvements over the previous implementation of this traffic simulator. Most of these improvements serve to improve on what data are accessible for the user, and the relative realism of real world traffic laws. Previously, we used randomness to simulate driving and car generation. This was and is a very easy thing for a developer to code, but in turn, limits how a user is able to learn from a simulation, as it disallows users from tinkering meaningfully with their simulation and only allows them to learn through observing the simulation during run time. The removal of random elements, while increasing workload for the designers and developers, does make for a more useful learning experience for users, as they can easily see how their decisions impact a simulation with the same inputs in other fields.

Quality of life changes like data-scraping tools provide insights on each simulation, allowing users to review what they've done with ease. Aside from that, changes for realism like road closures, vehicles of other speeds, and cars being able to turn were also added to help create a simulator that closer imitates reality.

Unfortunately, this design still has drawbacks. The most harrowing thing about this implementation is the sheer complexity and customizability of it all; a new user facing down all these options - from map structure to traffic light timings to the speed of cars relative to trucks - can easily be overwhelmed with what they interact with and how they can interact with it. This issue only increases if a user wants more fine-grain control over a non-trivial number of cars, as that becomes unmanageable on a large scale. Tooltips or a tutorial would greatly aid the user in understanding the software, but that's more of a user interface design issue.

A smaller but still considerable concern is the difference in computing power each implementation requires, and this implementation certainly requires more. Both versions should be easy to run given a smaller amount of cars and trucks; the current implementation however has a more complicated algorithm for each piece of the road and would take some measurable more time to run.