INTERNET OF THINGS-GROUP 4
TRAFFIC MANAGEMENT SYSTEM

STUDENT NAME :S MALINI

REGISTER NUMBER:822621106008

NAAN MUDHALVAN I'D: au822621106008

COLLEGE NAME: ARIFA INSTITUTE OF TECHNOLOGY

COLLEGE CODE: 8226

EMAIL I'D:malinivani03@gmail.com

Introduction

The rapid speed at which urban growth is proceeding is the primary cause of the increasing traffic congestion on city roads. Because of this, vehicles can be standing for a long time. Long-term standing affects the environment in the form of vehicle pollution, ‘’"""""’’’’’’which causes human health issues related to breathing and delays in emergency situations such as accidents that may cause death. Stopping development to reduce traffic congestion may not be the solution; there are many other factors, apart from development, that contribute to traffic congestion. One of the factors is the increased number of vehicles, which can be worked on. So, it is very important to develop an intelligent system that can be used to reduce traffic congestion by addressing the number of vehicles. Nowadays, various types of technologies for advancement are being developed. These include the Internet of Things (IoT), machine learning (ML), microcontrollers, wireless sensor networks (WSNS), and fuzzy logic (FL), which are used to better control traffic in complex situations

Image acquisition:

A digital image acquisition represents features of the vehicle in the form of pixels, which are extracted from a real-time scene. The main aspect of this component is to deal with the captured vehicles before evaluating them. This component describes how to obtain traffic conditions and from where it is obtained.

Utilization of static and dynamic attributes:

This component is responsible for extracting both static and dynamic vehicle attributes after receiving the captured scene in real-time. The speed of the vehicle, its trajectory, and its direction are examples of the vehicle’s dynamic attributes as they change over time. Static attributes of a vehicle include features such as the logo, color, type, license plate number, and other details that remain constant throughout time. This component explains how to use extracted attributes for developing ITMS.

Vehicle behavioral understanding:

This component is responsible for providing an understanding of internal and external factors in vehicles via the use of static and dynamic attributes obtained from cameras placed at various points along the road network.

Traffic software applications in ITMS:

In the field of ITMS, traffic software applications typically make use of cutting-edge technologies such as global positioning systems (GPS), traffic sensors, and real-time traffic data in order to collect and analyze information regarding the traffic situation on the roads. They are able to offer drivers real-time traffic updates, propose alternate routes, and assist in the management of emergencies such as accidents and road closures based on the information that they have access to.

ITMS applications:

This component provides a brief description of the many ITMSrelated applications, including electronic toll collection, environmental impact evaluation, TSCSs, security monitoring, anomaly detection, and illegal activity identification.

Traffic Scene Regions for Image Acquisition

ITMS is primarily used in the management of traffic in four distinct regions of traffic scenes by using imaging technology. The regions of the traffic scene are mentioned below.

• City Tunnel:

A city tunnel is a completely enclosed underground passageway with entry and exit points at either end. The darkness that surrounds the city tunnel during the day is similar to the darkness during the night. Therefore, during the ITMS development, there are lighting challenges in the city tunnel. This is solved by providing extra light so that the ITMS system can work perfectly.

• Highway:

A highway is a multi-lane road. On an urban highway, vehicles move fast, so they appear for a short period in the camera's field of view. Therefore, imaging technologies should be advanced so that they are able to capture the scene at a fast frame rate, which can be used for vehicle detection, vehicle classification, and vehicle tracking. The vehicle speed capturing challenge is addressed by supplying a high frame rate video camera similar to that used in ITMS.

• Road intersection:

An intersection is a point at which two or more roads intersect. At the road intersection, vehicles usually move left, right, and round. As a result, plenty of poses occur, making it challenging to identify and track them. There are many vehicle detectors and methods, such as "You Only Look Once" (YOLO) [1] and the Kalman filter [2], which are used at the intersection to detect multiple poses as well as the occluded vehicles to calculate the traffic signal accurately. The main role of a traffic signal is to provide efficient timing for vehicles in each lane. It can be both dynamic and fixed in nature. The dynamic traffic signal provides the time for passing vehicles from an intersection on the basis of the number of vehicles, and the fixed signal provides the time on the basis of historical data that is related to the number of vehicles moving at a particular intersection.

• Road section:

Vehicles moving at a slower rate on the section of the road during peak hours, and in times of heavy traffic congestion, may even come to a complete stop. This results in a long vehicle queue. Therefore, an appropriate lane should be planned with a powerful ITMS along with a rerouting approach to avoid long queues.

Vehicle Tracking

One of the most important and active fields of research in the science of CV is multiobject tracking. It finds considerable application in robotic vision, surveillance systems, and other commercial applications, such as the synthesis of surveillance video synopses. Luoet al. [68] provided details regarding multi-object tracking. A single object has been tracked using probabilistic-based trackers such as the Kalman filter, particle filter, metaheuristic trackers, and deterministic trackers. The Kalman filter and the particle filter are two of the most popular tracking techniques.

The Kalman filter improves the accuracy and reliability of tracking significantly when vehicle motion is blocked by other objects, which can result in tracking failure [69]. Actually, this is a kind of linear quadratic estimation for estimating linear-quadratic functions based on a set of measurements taken throughout time for more efficient and stable tracking. The use of sensors to locate objects and determine distances does not always provide correct results. This is a deficiency, but sensor fusion helps to compensate for it by lowering the error rate. As a result of sensor fusion, Kim et al. [70] developed a strategy that makes use of an Extended Kalman Filter, which takes into account the methodologies that lidar and radar sensors use to compute distance. The distance-dependent properties of the lidar and radar sensors were analyzed, and a reliability function was developed in order to include the distance-dependent properties of the sensors in the Kalman filter. There are a number of issues that often arise during tracking, including the fact that sensors do not always provide correct readings and that the motion model of the target may shift as a result of changes in the environment or interactions with other targets. Using the concept of interactions, Khalkhali et al. [71] developed an approach known as an Interactive Kalman filter.

Vehicle Recognition

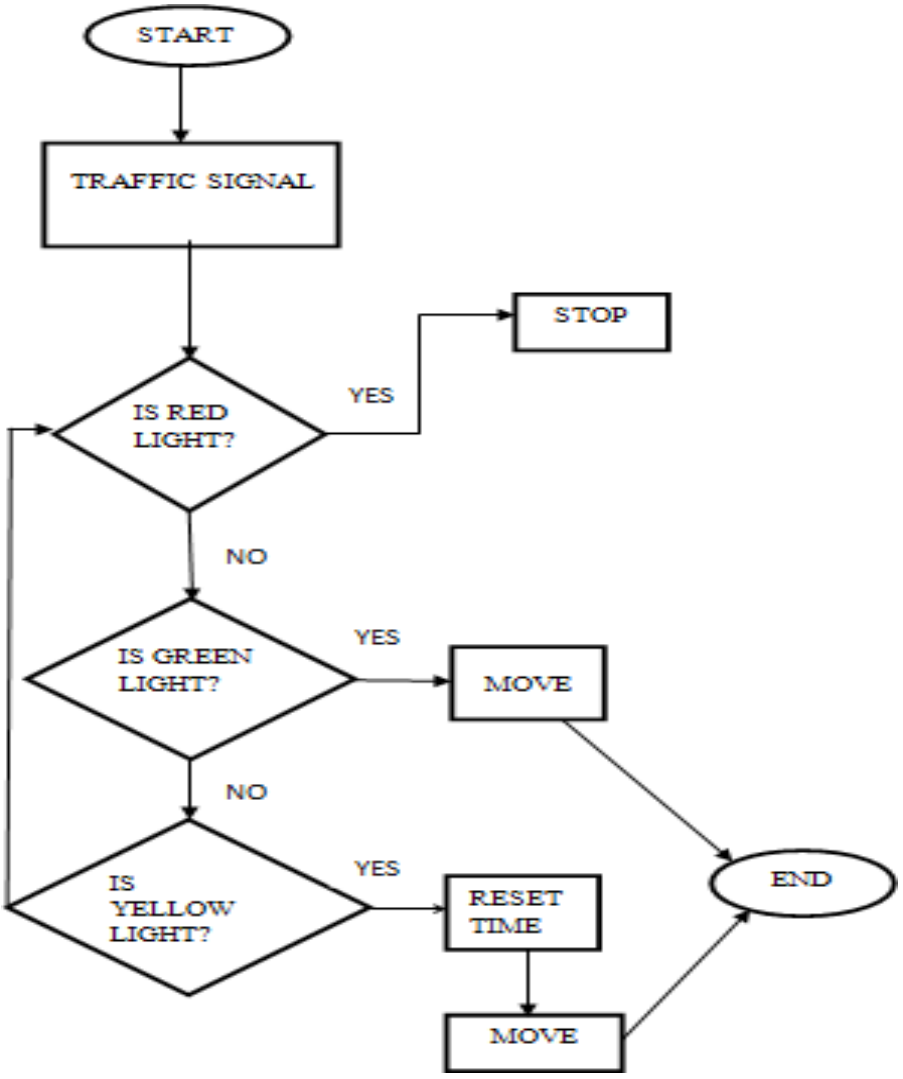
The process of identifying the types of vehicles that are present on the road is referred to as "vehicle recognition". The following section discusses the numerous vehicle recognition-based techniques that make use of vehicle color, vehicle logo, vehicle license plate numbers, vehicle shape, and appearance.

Traffic Signal Control Systems (TSCSS)

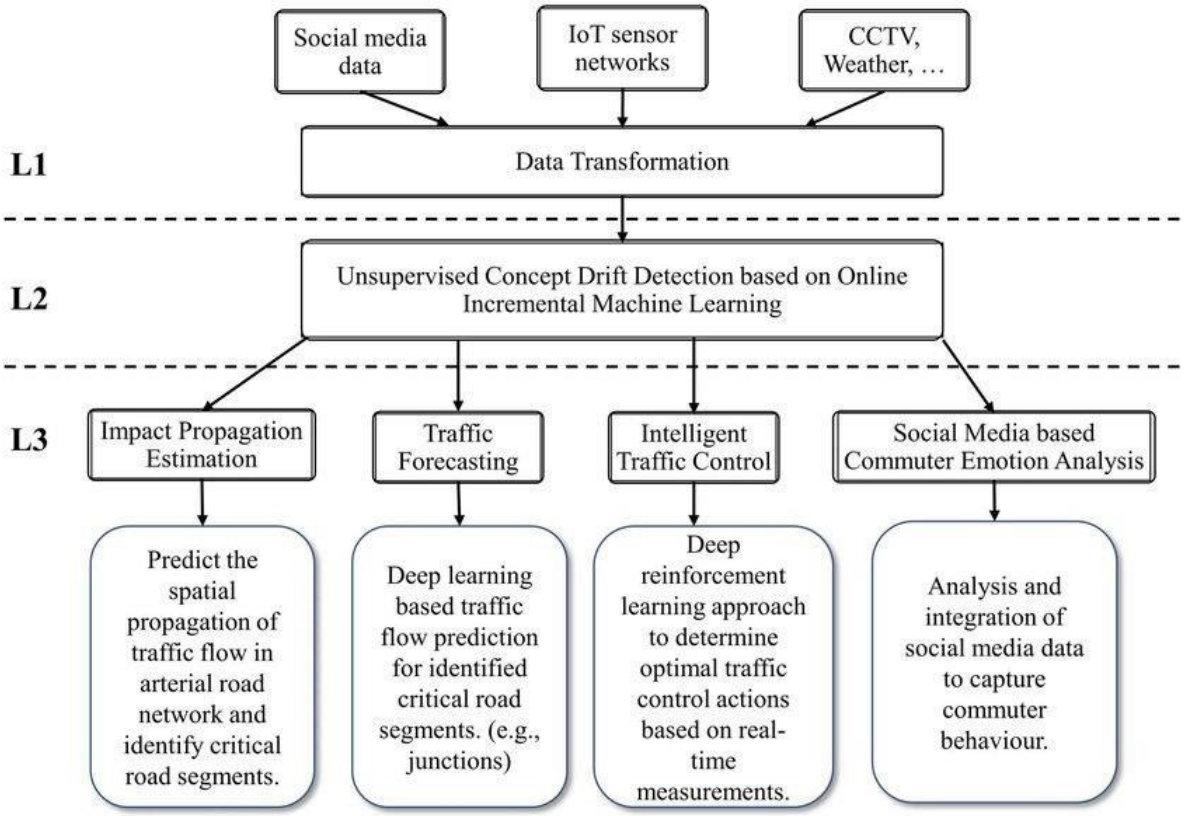
People are leaving their hometowns in search of places that provide greater employment opportunities and a higher quality of life than what they can find in their current locations. As a direct consequence of the fast urbanization that is taking place, cities are seeing growth in the total amount as well as the variety of traffic.

The problems caused by traffic are as follows:

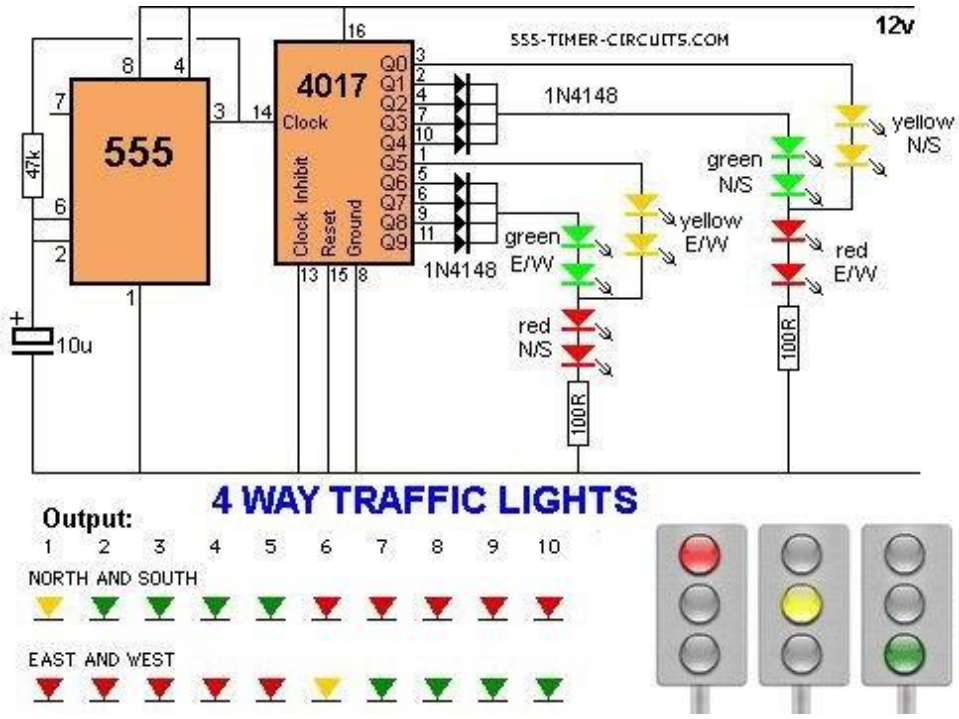
- Increases the total amount of travel time
- The use of fuel between intersection lines
- Increased contributions to the air pollution caused by emissions
- Unfortunate events;
- Managing an emergency is challenging.

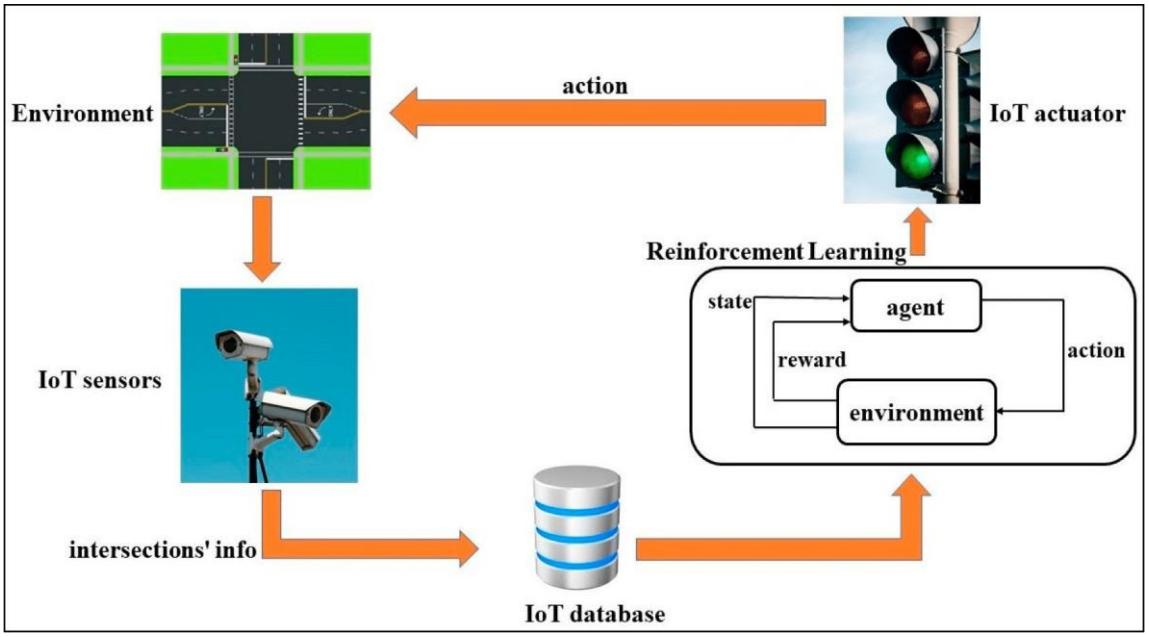


Traffic flowing to a junction from all the directions is most of the time unequal. This factor is not taken into account by the conventional traffic management system leading to unbalanced distribution of traffic from all directions which leads to air pollution and sever health issues, as there are around 67 lakhs on vehicles in single metropolitan city like Bangalore where people are facing this issue. A smart traffic management system is, where traffic is constrained by the administration framework, which controls the traffic lights as per the continuous circumstance of traffic moving from every extraordinary bearing in an intersection. This continuous information is gathered from different sensors set at equivalent interims of separation at an intersection. This information is gathered and brought to a control framework which independently ascertains the ideal time for the arrival of the green sign to every specific heading in an intersection so as to counteract traffic heaping up[1]. The subsequent significant issue which we are taking a gander at in this venture is about the development of crisis vehicles (fire engines, ambulances and police vehicles) in packed intersections of a city.



Schematics diagram:





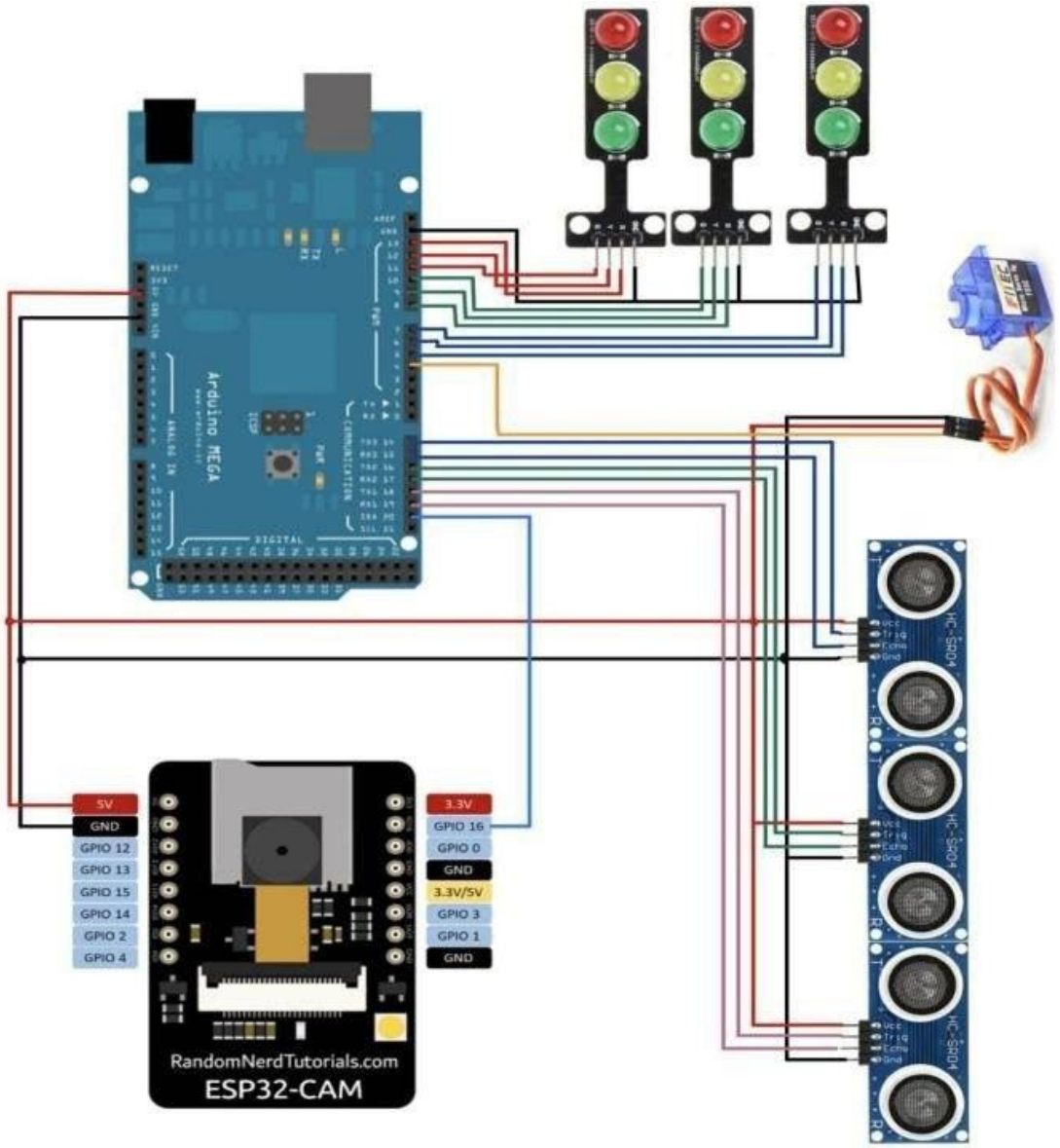
SOFTWARE REQUIRMENT:

Arduino IDE has been installed on the PC. In addition to a few other functions, the IDE includes a compiler. The Arduino language seems similar to C++ on the surface. Through the use of the IDE, the program is written, constructed, and uploaded to the board. The language can be understood easily. The communication port that the Arduino board is attached to, along with a variety of Arduino boards with different controllers, are both selectable in the IDE. Real-time analytics, machine learning, pervasive computing, inexpensive sensors, and embedded systems have all helped the internet of things grow.

Circuit diagram

This is system architecture is shown in figure 6, which is consists of Arduino mega, ultrasonic sensor, cam esp32 & servo motor, smart traffic light appliances that represented by (three streets, three-color lights (red, yellow, green)) which is shown in figure 6.

We have three lines have three lights (green, red, yellow), □□□ ultrasonic sensors detect a car violating a red traffic light then send the signal to the servo motor have cam esp32, each street we give angle the street number 1 0 angles, street number 2 90 angles, street number 3 180 angle servo motor will move with cam esp32 to that side a car violation a red traffic light to take picture car number and send it to our database by using telegram.



TRAFFIC SIMULATION:

```
from itertools import chain
from typing import List, Dict, Tuple, Set, Optional

from scipy.spatial import distance

from TrafficSimulator.road import Road
from TrafficSimulator.traffic_signal import TrafficSignal
from TrafficSimulator.vehicle_generator import VehicleGenerator
from TrafficSimulator.window import Window
```

```
class Simulation:
    def __init__(self, max_gen: int = None):
        self.t = 0.0  # Time
        self.dt = 1 / 60  # Time step
        self.roads: List[Road] = []
        self.generators: List[VehicleGenerator] = []
        self.traffic_signals: List[TrafficSignal] = []

        self.collision_detected: bool = False
        self.n_vehicles_generated: int = 0
        self.n_vehicles_on_map: int = 0

        self._gui: Optional[Window] = None

        self._non_empty_roads: Set[int] = set()
        # To calculate the number of vehicles in the junction, use:
        # n_vehicles_on_map - _inbound_roads vehicles - _outbound_roads vehicles
        self._inbound_roads: Set[int] = set()
        self._outbound_roads: Set[int] = set()

        self._intersections: Dict[int, Set[int]] = {}  # {Road index: [intersecting roads' indexes]}
        self.max_gen: Optional[int] = max_gen # Vehicle generation limit
        self._waiting_times_sum: float = 0  # for vehicles that completed the journey

    def add_intersections(self, intersections_dict: Dict[int, Set[int]]) -> None:
        self._intersections.update(intersections_dict)

    def add_road(self, start: Tuple[int, int], end: Tuple[int, int]) -> None:
        road = Road(start, end, index=len(self.roads))
        self.roads.append(road)

    def add_roads(self, roads: List[Tuple[int, int]]) -> None:
        for road in roads:
            self.add_road(*road)

    def add_generator(self, vehicle_rate, paths: List[List]) -> None:
        inbound_roads: List[Road] = [self.roads[roads[0]] for weight, roads in paths]
        inbound_dict: Dict[int: Road] = {road.index: road for road in inbound_roads}
        vehicle_generator = VehicleGenerator(vehicle_rate, paths, inbound_dict)
        self.generators.append(vehicle_generator)

        for (weight, roads) in paths:
            self._inbound_roads.add(roads[0])
            self._outbound_roads.add(roads[-1])

    def add_traffic_signal(self, roads: List[List[int]], cycle: List[Tuple],
                           slow_distance: float, slow_factor: float, stop_distance: float) -> None:
        roads: List[List[Road]] = [[self.roads[i] for i in road_group] for road_group in roads]
        traffic_signal = TrafficSignal(roads, cycle, slow_distance, slow_factor, stop_distance)
        self.traffic_signals.append(traffic_signal)

    @property
    def gui_closed(self) -> bool:
        """ Returns an indicator whether the GUI was closed """
```

```
    return self._gui and self._gui.closed

@property
def non_empty_roads(self) -> Set[int]:
    """ Returns a set of non-empty road indexes """
    return self._non_empty_roads

@property
def completed(self) -> bool:
    """
    Whether a terminal state (as defined under the MDP of the task) is reached.
    """
    if self.max_gen:
        return self.collision_detected or (self.n_vehicles_generated == self.max_gen
                                           and not self.n_vehicles_on_map)
    return self.collision_detected

@property
def intersections(self) -> Dict[int, Set[int]]:
    """
    Reduces the intersections' dict to non-empty roads
    :return: a dictionary of {non-empty road index: [non-empty intersecting roads indexes]}
    """
    output: Dict[int, Set[int]] = {}
    non_empty_roads: Set[int] = self._non_empty_roads
    for road in non_empty_roads:
        if road in self._intersections:
            intersecting_roads = self._intersections[road].intersection(non_empty_roads)
            if intersecting_roads:
                output[road] = intersecting_roads
    return output

@property
def current_average_wait_time(self) -> float:
    """ Returns the average wait time of vehicles
    that completed the journey and aren't on the map """

    on_map_wait_time = 0
    completed_wait_time = 0
    n_completed_journey = self.n_vehicles_generated - self.n_vehicles_on_map
    if n_completed_journey:
        completed_wait_time = round(self._waiting_times_sum / n_completed_journey, 2)
    if self.n_vehicles_on_map:
        total_on_map_wait_time = sum(vehicle.get_wait_time(self.t) for i in self.non_empty_roads
                                     for vehicle in self.roads[i].vehicles)
        on_map_wait_time = total_on_map_wait_time / self.n_vehicles_on_map
    return completed_wait_time + on_map_wait_time

@property
def inbound_roads(self) -> Set[int]:
    return self._inbound_roads

@property
def outbound_roads(self) -> Set[int]:
    return self._outbound_roads

def init_gui(self) -> None:
```

```
    """ Initializes the GUI and updates the display """
    if not self._gui:
        self._gui = Window(self)
    self._gui.update()

def run(self, action: Optional[int] = None) -> None:
    """ Performs n simulation updates. Terminates early upon completion or GUI closing
    :param action: an action from a reinforcement learning environment action space
    """
    n = 180 # 3 simulation seconds
    if action:
        self._update_signals()
        self._loop(n)
        if self.collision_detected or self.gui_closed:
            return
        self._update_signals()
        if self.completed or self.gui_closed:
            return
    self._loop(n)

def update(self) -> None:
    """ Updates the roads, generates vehicles, detect collisions and updates the gui """
    # Update every road
    for i in self._non_empty_roads:
        self.roads[i].update(self.dt, self.t)

    # Add vehicles
    for gen in self.generators:
        if self.max_gen and self.n_vehicles_generated == self.max_gen:
            break
        road_index = gen.update(self.t, self.n_vehicles_generated)
        if road_index is not None:
            self.n_vehicles_generated += 1
            self.n_vehicles_on_map += 1
            self._non_empty_roads.add(road_index)

    self._check_out_of_bounds_vehicles()

    self._detect_collisions()

    # Increment time
    self.t += self.dt

    # Update the display
    if self._gui:
        self._gui.update()

def _loop(self, n: int) -> None:
    """ Performs n simulation updates. Terminates early upon completion or GUI closing"""
    for _ in range(n):
        self.update()
        if self.completed or self.gui_closed:
            return

def _update_signals(self) -> None:
    """ Updates all the simulation traffic signals and updates the gui, if exists """
    for traffic_signal in self.traffic_signals:
```

```
            traffic_signal.update()
        if self._gui:
            self._gui.update()

    def _detect_collisions(self) -> None:
        """ Detects collisions by checking all non-empty intersecting vehicle paths.
        Updates the self.collision_detected attribute """
        radius = 3
        for main_road, intersecting_roads in self.intersections.items():
            vehicles = self.roads[main_road].vehicles
            intersecting_vehicles = chain.from_iterable(
                self.roads[i].vehicles for i in intersecting_roads)
            for vehicle in vehicles:
                for intersecting in intersecting_vehicles:
                    if distance.euclidean(vehicle.position, intersecting.position) < radius:
                        self.collision_detected = True
                        return

    def _check_out_of_bounds_vehicles(self):
        """ Check roads for out-of-bounds vehicles, updates self.non_empty_roads """
        new_non_empty_roads = set()
        new_empty_roads = set()
        for i in self._non_empty_roads:
            road = self.roads[i]
            lead = road.vehicles[0]
            # If first vehicle is out of road bounds
            if lead.x >= road.length:
                # If vehicle has a next road
                if lead.current_road_index + 1 < len(lead.path):
                    # Remove it from its road
                    road.vehicles.popleft()
                    # Reset the position relative to the road
                    lead.x = 0
                    # Add it to the next road
                    lead.current_road_index += 1
                    next_road_index = lead.path[lead.current_road_index]
                    new_non_empty_roads.add(next_road_index)
                    self.roads[next_road_index].vehicles.append(lead)
                    # road.vehicles.popleft()
                    if not road.vehicles:
                        new_empty_roads.add(road.index)
                else:
                    # Remove it from its road
                    road.vehicles.popleft()
                    # Remove from non_empty_roads if it has no vehicles
                    if not road.vehicles:
                        new_empty_roads.add(road.index)
                    self.n_vehicles_on_map -= 1
                    # Update the waiting times sum
                    self._waiting_times_sum += lead.get_wait_time(self.t)

        self._non_empty_roads.difference_update(new_empty_roads)
        self._non_empty_roads.update(new_non_empty_roads)
```