ADAPTIVE TRAFFIC MANAGEMENT SYSTEM

A MINI-PROJECT REPORT

submitted by

DAN SHIJU [SCM21CS042]

HARIGOVIND B [SCM21CS060]

FIDHA FATHIMA SALEEM [SCM21CS051]

JINO JOHNSON [SCM21CS068]

to

The APJ Abdul Kalam Technological University

in partial completion of the criteria for the degree award

of

Bachelor of Technology in

Computer Science and Engineering

Department of Computer Science and Engineering
SCMS SCHOOL OF ENGINEERING AND TECHNOLOGY
(*Affiliated to APJ Abdul Kalam Technological University*)
VIDYA NAGAR, PALISSERY, KARUKUTTY
ERNAKULAM – 683582

MAY 2024

DECLARATION

We undersigned hereby declare that the project report ADAPTIVE TRAFFIC MANAGEMENT SYSTEM (ATMS), submitted for partial fulfilment of the requirements for the award of the degree of Bachelor of Technology of the APJ Abdul Kalam Technological University, Kerala is a bona fide work done by me under the supervision of Ms Asha S. This submission represents our ideas in our own words and where ideas or words of others have been included, we have adequately and accurately cited and referenced the sources. We also declare that we have adhered to the ethics of academic honesty and integrity and have not misrepresented or fabricated any data or idea or fact or source in my submission. We understand that any violation of the above will be a cause for disciplinary action by the institute and/or the University and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been obtained. This report has not previously formed the basis for the award of any degree, diploma or similar title of any other University.

Place: Ernakulam Signature

Date: 6 May 2024

Name of the students

DAN SHIJU

HARIGOVIND B

FIDHA FATHIMA SALEEM

JINO JOHNSON

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING SCMS SCHOOL OF ENGINEERING

AND TECHNOLOGY Vidya Nagar, Karukutty,

Kerala – 683 576

CERTIFICATE

This is to certify that the report entitled ADAPTIVE TRAFFIC MANAGEMENT SYSTEM, applying concepts of IOT and Web Development technologies, submitted by **DAN SHIJU**, **HARIGOVIND B, FIDHA FATHIMA SALEEM and JINO JOHNSON** to the APJ Abdul Kalam Technological University in partial fulfilment of the requirements for the award of the Degree of Bachelor of Technology in (Computer Science and Engineering) is a bona fide record of the project work carried out by him/her under my/our guidance and supervision. This report in any form has not been submitted to any other University or Institute for any purpose.

Project Guide	Project Coordinator	Head of the Department
Ms. ASHA S	**Mr. Mahesh K M**	**Dr. Manish T I**
Assistant Professor	Assistant Professor	Professor
Department of CO	Department of CSE	Department of CSE
SSET	SSET	SSET

ACKNOWLEDGEMENT

We are greatly indebted to **Dr. Anitha G.Pillai**, Principal, SCMS School of Engineering and Technology, Ernakulam and **Dr. Manish T I**, Professor and Head of Department, Department of Computer Science and Engineering, SCMS School of Engineering and Technology, who wholeheartedly permitted us to conduct this project.

We would like to thank our guide, **Ms. Asha S**, Assistant Professor, Department of Computer Science and Engineering, SCMS School of Engineering and Technology who gave us valuable guidance. We sincerely thank our Class coordinator Mahesh K M, Assistant Professor, Department of Computer Science and Engineering, SSET, who has given us valuable guidance and has supported us all the way. We would like to express sincere gratitude to all the teachers of the Department of Computer Science who gave us moral and technical support. We would like to thank the supporting staff in the computer lab whose dedicated work kept the lab working smoothly thus enabling us to have access to various resources which helped us understand more about the project topic. We would also like to extend our gratitude to all friends and family for providing us with the necessary resources and support.

ABSTRACT

Traffic congestion at intersections significantly disrupts traffic flow and wastes motorist time. Traditional traffic light systems with fixed timings struggle to adapt to dynamic traffic patterns. This system proposes an improved system for adaptive traffic signal control that utilizes real-time vehicle detection and image processing techniques.

The system leverages object detection models, like YOLO, trained on vehicle datasets to identify and count vehicles within camera captured images at intersections. This real-time traffic data is then analysed to determine traffic density. Based on the density classification (low or high), the system dynamically adjusts the green light duration for each lane using a predefined formula. This approach optimizes traffic flow by allocating green light time efficiently according to the current demand.

The methodology also incorporates traffic simulation using Pygame for visualization and interactivity. This allows for testing the system's performance in various traffic scenarios before real-world deployment. Finally, real-world implementation involves camera installation for image capture and extensive testing to ensure functionality, accuracy, and effectiveness under diverse traffic conditions.

This improved system offers a promising solution for smart traffic management by dynamically adapting traffic signals based on real-time traffic data, ultimately reducing congestion and improving overall intersection performance.

CONTENTS

Contents **PageNo.**

LIST OF FIGURES

ABBREVIATIONS

ATMS	Adaptive Traffic Management System
YOLO	You Only Look Once
MATLAB	Matrix Laboratory
CCTV	Closed Circuit Television
OpenCV	Open Source Computer Vision Library
USB	Universal Serial Bus.
ROI	Return on Investmen

CHAPTER 1

INTRODUCTION

1.1 GENERAL BACKGROUND

Traffic congestion at junctions is a pervasive issue in urban areas worldwide, leading to wasted time, increased fuel consumption, and environmental pollution. Conventional traffic light control systems, reliant on fixed timing mechanisms, often fail to adapt to fluctuating traffic conditions, exacerbating congestion and frustration among commuters. As urban populations continue to grow, the demand for efficient traffic management solutions becomes increasingly pressing. In response, there has been a growing interest in the development of adaptive traffic signal control systems capable of dynamically adjusting signal timings based on real-time traffic data. Leveraging advancements in computer vision, machine learning, and simulation technologies, researchers have sought to design and implement innovative approaches to optimize traffic flow and alleviate congestion at intersections. This project builds upon existing research efforts by proposing a real-time adaptive traffic signal control system that integrates vehicle detection, traffic flow analysis, and simulation techniques to enhance junction performance and improve overall traffic management efficiency.

1.2 OBJECTIVE

- To develop a real-time adaptive traffic signal control system capable of dynamically adjusting signal timings based on observed traffic density at junctions.
- To optimize traffic flow.
- To reduce traffic conjunction.
- To save the waiting time of drivers in traffic.
- Validate system functionality and performance using simulation.

1.3 SCOPE OF PROJECT

The Adaptive Traffic Management System (ATMS) focuses on the development and testing of a prototype adaptive traffic signal control system utilizing real-time vehicle detection and image processing techniques. The following functionalities are included within the scope:

❖ **Vehicle Detection and Analysis:**

➢ Implementation of an object detection model (e.g., YOLO) trained on vehicle datasets to identify and count vehicles within the input images.

❖ **Traffic Density Analysis:**

➢ Development of algorithms to retrieve and find traffic density at the intersection.

❖ **Dynamic Green Light Allocation:**

➢ Design and implementation of a system to dynamically adjust the green light duration for each lane based on the classified traffic density. This includes the development and application of a predefined formula for green light allocation.

❖ **Traffic Simulation:**

➢ Creation of a traffic simulation environment using Pygame to evaluate system performance under various traffic scenarios before real-world deployment.

❖ **Testing and Validation:**

➢ Implementation of a comprehensive testing plan to assess the functionality, accuracy, and performance of the system under diverse real-world traffic conditions through limited camera installation and data collection.

Important Exclusions:

- Development or installation of physical traffic light infrastructure.
- Large-scale deployment and integration with existing traffic management networks.

This defined scope ensures a focused project that delivers a functional prototype of an adaptive traffic signal control system with the potential for future real-world applications.

CHAPTER 2

LITERATURE SURVEY

Kushi et al. [1] presents a traffic control system using image processing in MATLAB to adjust green, amber, and red-light timings based on traffic density. Two Arduino UNO boards manage light signals. Images captured by a webcam are processed in MATLAB to analyse road traffic density. Grayscale and binary image conversion simplify processing. Traffic density is calculated using object area ratios. MATLAB controls green and amber lights, while Arduino controls red lights. Communication between MATLAB and Arduino is via USB. The system offers dynamic traffic management, optimizing signal timings for improved traffic flow.

S.Chidhambaram et al. [2] proposed traffic control framework aims to enhance traditional systems by implementing several key strategies. Image acquisition is a critical component of the system, where captured images are continuously fed to the controller and compared with a reference image of an empty road.

Grayscale conversion is employed to simplify image analysis, focusing on luminance values to detect vehicles effectively. Canny edge detection, a widely utilized tool in image processing, further enhances boundary recognition while suppressing noise. By comparing pixel intensities and identifying discontinuities in brightness, the edge detector accurately detects vehicle boundaries and removes background noise.

Furthermore, the grayscale images undergo inversion to improve recognition, with the edge detector effectively suppressing background noise. Through comparison with the reference image, the system calculates traffic density for each lane, displaying the results on the MATLAB command window. Additionally, a sound sensor is integrated to identify emergency vehicles, ensuring they are given priority over lanes with higher traffic density. This comprehensive approach enables real-time monitoring and adjustment of traffic signal timings based on observed traffic density, ultimately optimizing traffic flow and reducing congestion at intersections.

Gandhi et al. (2022) in [3] leverages image processing and object detection (YOLO) to analyse camera footage (CCTV) from traffic junctions. It detects the number of vehicles within each class (car, bike, bus, truck) to calculate overall traffic density. This density information, along with potentially other factors, is then used by a signal switching algorithm to dynamically adjust the green

light duration for each lane. Red light times are updated correspondingly. Notably, the system maintains a minimum and maximum green light duration to prevent any lane from experiencing excessive wait times.

The paper highlights that unlike some proposals, this system prioritizes a familiar cyclic switching pattern, similar to existing systems. This avoids confusion for drivers accustomed to the current infrastructure and minimizes the need for behavioural adjustments.

Finally, the document mentions simulation results demonstrating a significant improvement (around 23%) in the number of vehicles passing through the intersection compared to the current static system. This suggests the proposed system's effectiveness in optimizing traffic flow.

2.1 FEASIBILITY STUDY

- **Technical Feasibility**
    - Technology Availability: Object detection models like YOLO and image processing libraries (OpenCV) are readily available.
    - Technical Expertise: Expertise in image processing, computer vision, and traffic control systems is required. However, this expertise can be obtained through hiring qualified personnel or collaboration with research institutions.

- **Economic Feasibility**
    - Costs: The primary costs include camera installation (if CCTV are not present), computing hardware for processing images, and software development.
    - Benefits: Potential benefits include reduced fuel consumption, improved travel times, and decreased emissions. A cost-benefit analysis should be conducted to assess the project's economic viability.

- **Financial Feasibility**
    - Funding Sources: Potential funding sources include government grants, public-private partnerships, or private investments.
    - Return on Investment (ROI): The ROI will depend on factors like traffic reduction achieved and potential cost savings.

- **Legal Feasibility**
    - Regulations: Compliance with traffic control regulations and data privacy laws is

essential.
    - Permits: Permits for camera installation and data collection may be required depending on local regulations.

- Market Feasibility
    - Market Need: Traffic congestion is a significant problem, making this system highly relevant.
    - Target Market: Cities, municipalities, and transportation authorities are the primary target market.

- Operational Feasibility
    - Implementation: Integration with existing traffic infrastructure may require collaboration with relevant authorities.
    - Maintenance: Regular system maintenance and potential software updates will be necessary.

This feasibility study suggests that developing ATMS is technically feasible. Further economic and financial analysis is required to assess its financial viability. Market demand and operational considerations also support its potential value. Overall, the project holds promise for improving traffic flow and reducing congestion in urban environments.

CHAPTER 3

EXISTING AND PROPOSED SYSTEM

3.1 EXISTING SYSTEM

Traffic congestion is a major challenge in urban areas, and various methods have been developed to manage traffic flow. Here's an overview of three commonly used systems:

1. Manual Traffic Control:

- **Description:** This traditional method relies on human intervention by traffic police officers.
- **Functionality:** Police officers use hand signals, whistles, and signage to direct traffic flow at intersections.
- **Limitations:** This approach is labor-intensive, prone to human error, and cannot effectively adapt to fluctuating traffic patterns. It's generally only used in specific situations, such as during construction or accidents.

2. Conventional Traffic Lights with Static Timers:

- **Description:** These are the most common type of traffic control system, relying on pre-programmed timings.
- **Functionality:** Traffic lights cycle through red, yellow, and green phases based on fixed time intervals set for each lane or direction.
- **Limitations:** Static timings cannot adapt to real-time traffic conditions. During low traffic periods, unnecessary delays occur. Conversely, during peak hours, fixed timings can lead to congestion. This inflexibility reduces overall traffic flow efficiency.

3. Traffic Control with Electronic Sensors:

- **Description:** This method incorporates sensors embedded in the road surface, such as loop detectors or radar sensors.
- **Functionality:** Sensors collect data on traffic presence and volume. This data is then used to dynamically adjust traffic light timings based on real-time traffic conditions.
- **Limitations:** While offering some improvement over static timers, sensor-based systems may not always be able to handle complex traffic patterns effectively. Additionally, installation and maintenance of these sensors can be costly.

3.2 PROPOSED SYSTEM

The proposed ATMS aims to overcome the limitations of existing traffic control systems by introducing adaptive and responsive features. The ATMS will utilize Python, YOLO object detection, OpenCV, and Pygame to create a dynamic traffic management solution. The key features of the proposed ATMS include:

- Real-time Traffic Monitoring: The system will utilize image processing techniques to monitor traffic conditions in real-time. Using YOLO object detection and OpenCV, the ATMS will accurately detect vehicles and pedestrians on the road.
- Dynamic Traffic Signal Adjustment: Based on the real-time traffic data collected, the ATMS will dynamically adjust traffic signals to optimize traffic flow. By analyzing traffic density, vehicle speed, and other relevant factors, the system will intelligently manage signal timings to reduce congestion and improve traffic efficiency.
- Dynamic Feedback Loop: The proposed system will incorporate a dynamic feedback loop to continuously adapt to changing traffic conditions. By collecting and analyzing real-time data, the ATMS will adjust traffic signal timings and control policies to optimize traffic flow.

By leveraging the capabilities of Python, YOLO, OpenCV, and Pygame, the ATMS will provide a flexible and efficient solution for managing traffic flow. The system's adaptability and responsiveness will help reduce congestion, improve safety, and enhance overall traffic efficiency on the roads. In this project, we will design, implement, and evaluate the ATMS, aiming to create a more resilient and effective traffic management system for urban environments. Through this innovative solution, we aim to contribute to the development of smarter and more sustainable urban transportation systems.

3.2.1 ADVANTAGES

- Traffic signals change automatically
- Adaptive to changing conditions
- Scalable for various locations
- Cost required is less
- Optimizes traffic flow

3.3 METHODOLOGY

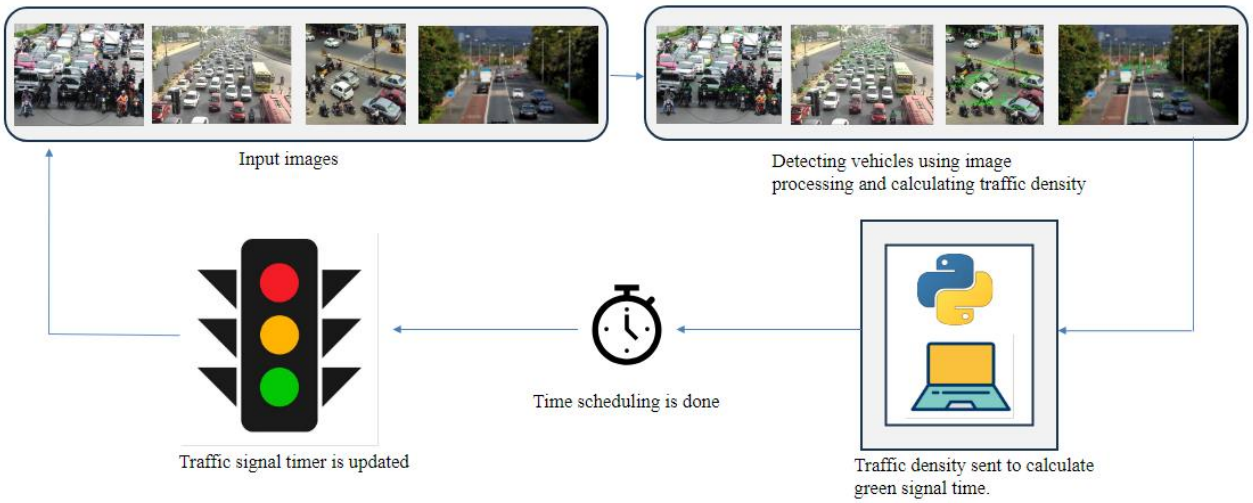

Fig 3.1. Proposed System Model

The proposed model is show in Fig 3.1. The methodology commences with vehicle detection, a critical step in the process. A robust object detection model, such as YOLO, is utilized for this purpose. Trained on extensive vehicle detection datasets, YOLO is adept at accurately identifying vehicles in input images. Once applied to the input image, YOLO detects vehicles and delineates bounding boxes around them, enabling precise identification. The total number of bounding boxes is then tallied to ascertain the overall count of vehicles present in the image.

Subsequently, traffic flow analysis ensues, leveraging OpenCV, a powerful library for computer vision tasks. OpenCV facilitates the determination of vehicle density in various areas of the image, including lanes and intersections. By processing the image using OpenCV algorithms, such as contour detection and region of interest masking, the density of vehicles in different regions is calculated. This analysis serves as a foundational element for the subsequent adjustment of green light durations.

The adjustment of green light phases is a pivotal aspect of the methodology, aimed at optimizing traffic flow. Utilizing the traffic density information obtained from the previous step, the duration of the green light phase for each lane is dynamically adjusted. This adjustment is computed using a formula that takes into account the number of vehicles, average time, and the number of lanes. By dynamically adapting green light durations based on real-time traffic conditions, the system aims to alleviate congestion and enhance traffic efficiency.

Following this, traffic simulation is conducted to visualize and interact with the traffic management scenario. Pygame , coupled with OpenCV, facilitates the creation of a dynamic simulation environment. This environment provides a visual representation of road networks, intersections, and vehicle movement, allowing for interactive exploration and analysis of traffic flow dynamics.

Finally, implementation and testing are to be carried out to validate the functionality, accuracy, and performance of the system in real-world scenarios. Images are to be collected through cameras to emulate real-world conditions, and extensive testing is conducted to evaluate the system's effectiveness in optimizing traffic flow and reducing congestion under various traffic conditions. Through rigorous testing and validation, the system aims to demonstrate its capability to enhance traffic management and improve overall junction performance.

CHAPTER 4

IMPLEMENTATION DETAILS

4.1 SYSTEM REQUIREMENT SPECIFICATION

4.1.1 SOFTWARE REQUIREMENTS

- Operating System: Windows 11
- Language: Python (3.11.5)
- Object recognition algorithm: YOLOv3
- Library: OpenCV (4.9.0), Pygame (2.5.2)

4.2 PLATFORM AND IDE

4.2.1 Visual Studio Code

Visual Studio is a powerful developer tool that you can use to complete the entire development cycle in one place. It is a comprehensive integrated development environment (IDE) that you can use to write, edit, debug, and build code, and then deploy your app.

Beyond code editing and debugging, Visual Studio includes compilers, code completion tools, source control, extensions, and many more features to enhance every stage of the software development process.

You can compile and build your applications to create builds immediately and test them in a debugger. Visual Studio also provides several options that you can configure when building applications. You can create a custom build configuration in addition to the built-in configurations, hide certain warning messages, or increase build output information.

4.2.2 Front End Implementation

The frontend implementation of the project focuses on creating an intuitive and interactive user interface for users to interact with the traffic control system. This interface will be developed using Pygame 2.5.2 to provide a visually appealing simulation environment. Users will be able to observe traffic flow dynamics, adjust parameters, and analyze simulation results in real-time through the graphical representation of road networks, intersections, traffic lights, and vehicles. The frontend will allow users to input parameters such as traffic density and signal timings, and visualize the effects of these changes on traffic flow. It will also include interactive features such as sliders and buttons to

enable users to control the simulation and explore different scenarios.

4.2.3 Back End Implementation

The backend implementation of the project focuses on the computational aspects of the traffic control system, including vehicle detection, traffic flow analysis, and signal optimization. This will be implemented using OpenCV and Python programming language. The backend will utilize a robust object detection model like YOLO for vehicle detection, enabling precise identification of vehicles in input images. OpenCV will be leveraged for traffic flow analysis, allowing for the determination of vehicle density in different areas of the image. The backend will dynamically adjust green light durations based on real-time traffic conditions, utilizing the traffic density information obtained from traffic flow analysis. Additionally, the backend will handle the simulation environment's logic, including the interaction between the graphical representation and the underlying traffic control algorithms. Through efficient computation and integration of algorithms, the backend will ensure the seamless operation of the traffic control system and enable effective traffic management in real-world scenarios.

4.3 APPLICATION DEVELOPMENT AND IMPLEMENTATION

The indicated system has proposed and developed an adaptive traffic control system. The methodology and processes are designed and created to provide a useful and effective solution in controlling and scheduling traffic in urban areas. The proposed system performed better in finding the traffic density from images. This is because the system uses YOLO object recognition algorithm and OpenCV.

The communication has been done through the use of inheritance between Vehicle_Detection and Simulation class. The vehicle count data and the order of traffic signal is displayed continuously for the verification of proper working of the system.

The simulation was made to visualize and interact with the traffic management scenario. It helps anyone to understand the working of the system.

CHAPTER 5

SYSTEM DESIGN

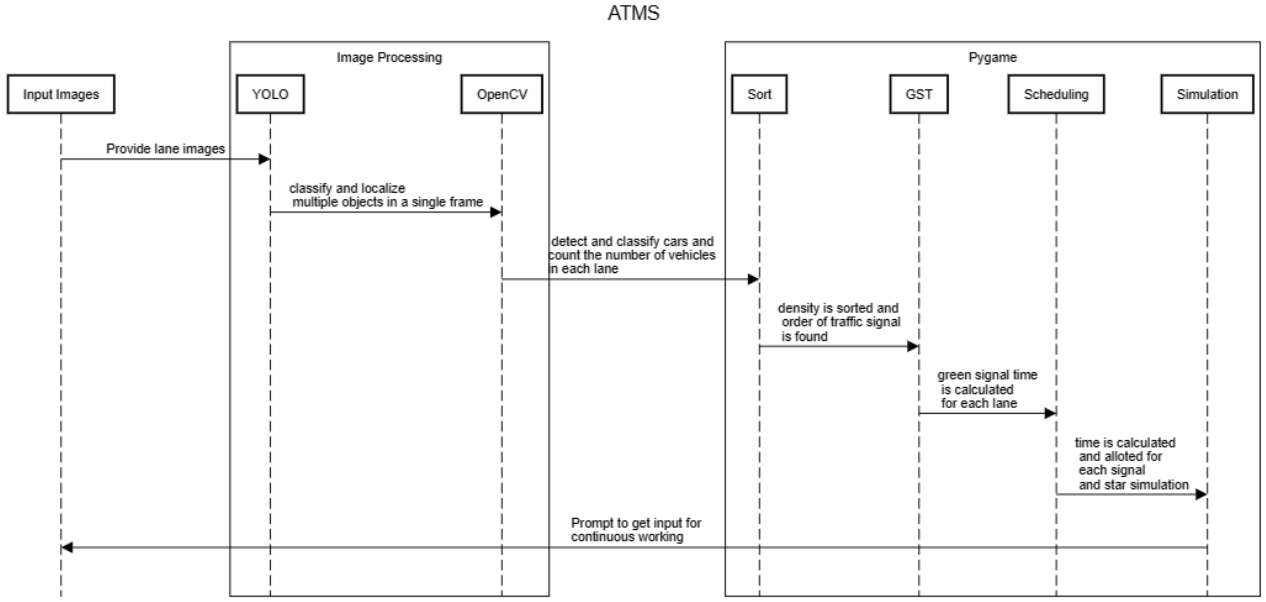

Fig 5.1. Sequence Diagram

ATMS tackles congestion through a sophisticated blend of real-time traffic monitoring and dynamic traffic light scheduling. The sequence diagram is shown in Fig 5.1. At the core lies YOLO, a powerful object detection algorithm. YOLO acts as the system's eyes, identifying and precisely locating vehicles within each lane provided in the input images. This information is then passed to OpenCV, a library specifically designed for computer vision tasks. OpenCV acts as the image processing engine, efficiently preparing the visual data for further analysis. It count the number of vehicles in each image and store in a dictionary with key as image file name and values as count.

The system then takes a crucial step by intelligently sorting the data based on the detected vehicle density. The order of the traffic signal is found based on the sorted density. The sort plays a central role in orchestrating the traffic signals based on the calculated density. From the count, the Green Signal Time (GST) determines the optimal green light time for each lane, ensuring a smooth flow of traffic. Now we have all the data required for scheduling the time to its appropriate traffic signals. The time of red and yellow light is calculated based on the GST and assigned to the respective traffic light.

Pygame, a graphics library, then steps onto the stage. Pygame allows the system to simulate the traffic lights according to the calculated green timings. This simulation acts as a virtual test track, providing valuable insights into how the real-world traffic flow might respond to the adjustments. Essentially, it's a way to test the efficacy of the calculated light timings before they're implemented in the real world.

Finally, after each traffic signal duration ends the system prompts input. This input makes the system adaptable and allows for continuous operation based on user discretion. By combining these elements, the ATMS design offers a data-driven approach to optimizing traffic light timing and reducing congestion. It's a comprehensive system that leverages cutting-edge computer vision techniques, real-time data analysis, and strategic scheduling to create a more efficient traffic management solution.

CHAPTER 6

EXPERIMENTAL RESULTS

The Adaptive Traffic Management System (ATMS) project was successfully completed on time and thoroughly tested with real-time traffic data. By leveraging Python, YOLO object detection, OpenCV, and Pygame, the ATMS demonstrated its capability to dynamically adjust traffic signals and optimize traffic flow. The project aims to improve overall traffic flow by providing precise and accurate readings of traffic conditions. Users can benefit from reduced congestion, safer roads, and more efficient transportation.

6.1 RESULT

Input Images

The four images that represents vehicles waiting at an intersection is provided.

Fig 6.1. img1.jpeg

Fig 6.2. img2.jpeg

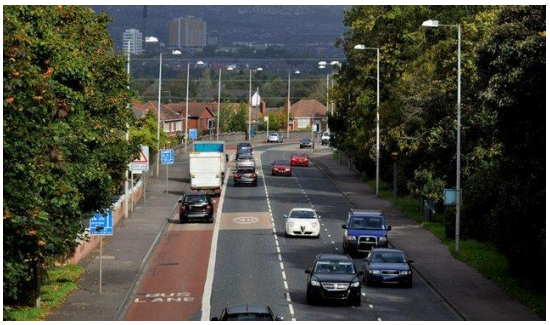

Fig 6.3. img3.jpeg

Fig 6.4. img4.jpeg

Processed Images

The input images are detected by YOLO. It identifies each object and OpenCV classify and assign labels and boxes around detected object

Fig 6.5. img1.jpeg

Fig 6.6. img2.jpeg

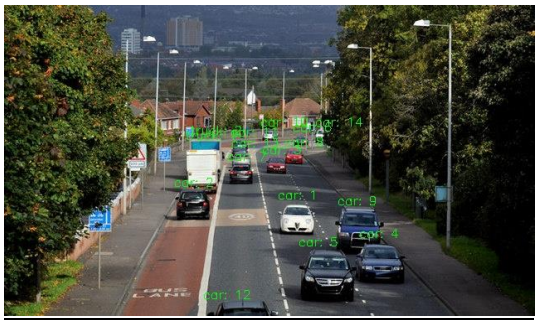

Fig 6.7. img3.jpeg

Fig 6.8. img4.jpeg

The count of vehicles in each image is found displayed.
[3,0,1,2] – The order of traffic light (Traffic light 1 is 0 – Traffic light 4 is 3)
{0:36, 1:21, 2:16, 3:44} – The GST allotted for the respective traffic signal

```
PS D:\VisualStudioPython\ATMS> python -u "d:\VisualStudioPython\ATMS\Simulation\simulation.py"
pygame 2.5.2 (SDL 2.28.3, Python 3.11.5)
Hello from the pygame community. https://www.pygame.org/contribute.html
[ WARN:0@1.587] global net_impl.cpp:178 cv::dnn::dnn4_v20231225::Net::Impl::setUpNet DNN module was not built with CUDA backend; switching to CPU
Counts for img1.jpg:, Total count: 36
Counts for img2.jpg:, Total count: 21
Counts for img3.jpg:, Total count: 16
Counts for img4.jpg:, Total count: 44
[3, 0, 1, 2]
{0: 36, 1: 21, 2: 16, 3: 44}
PS D:\VisualStudioPython\ATMS>
```

Fig 6.9. Count of each image, the order of traffic signal and green time allotted for each lane

Simulation

Traffic light arranged as:

- Traffic Light 1 - Top Left
- Traffic Light 2 - Top Right
- Traffic Light 3 - Bottom Right
- Traffic Light 4 - Bottom Left

As per the order [3,0,1,2] Traffic light 4 get started with GST = 44

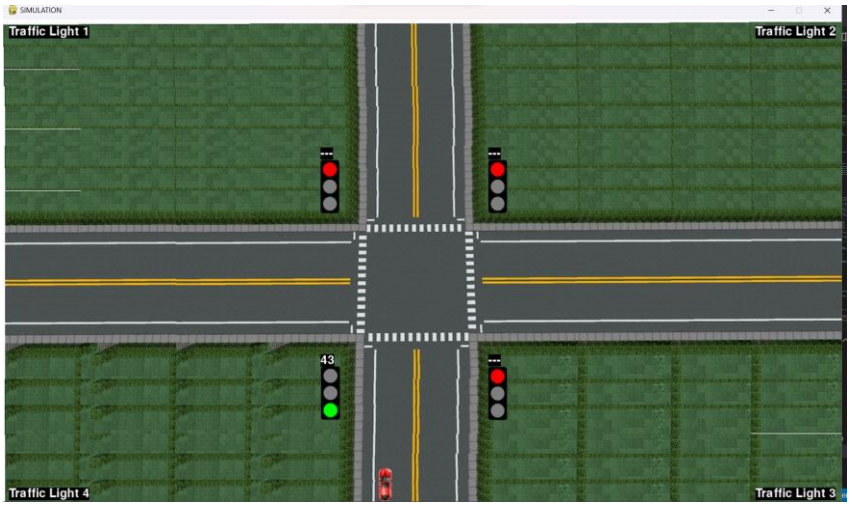

Fig 6.10. Traffic light 4

Then Traffic light 1 works with GST = 35

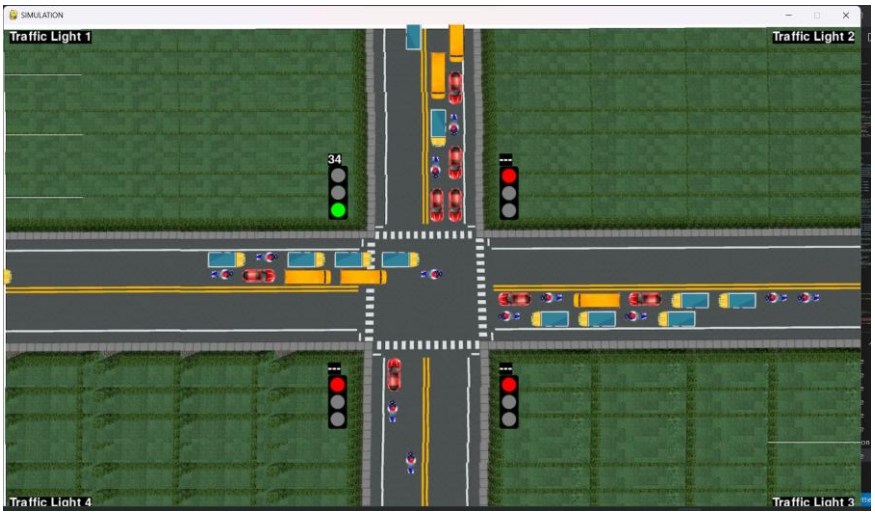

Fig 6.11. Traffic light 1

The simultaneous working of green and red light between traffic light

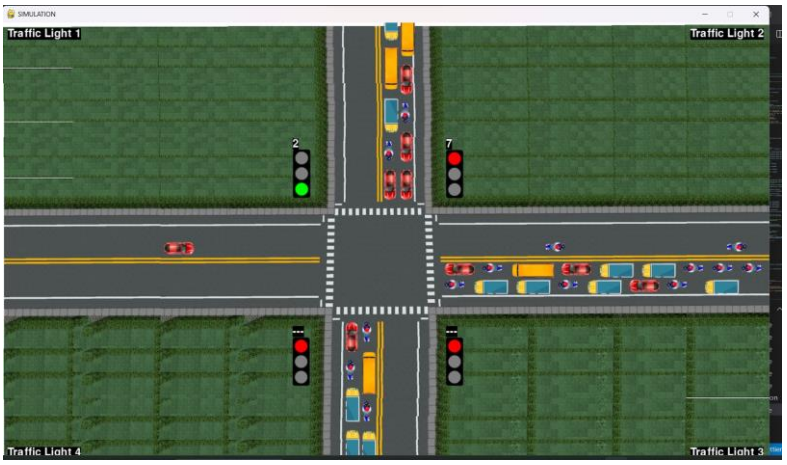

Fig 6.12. Traffic light 2 is going to get activated after 7 seconds

The simultaneous working of yellow and red light between traffic light

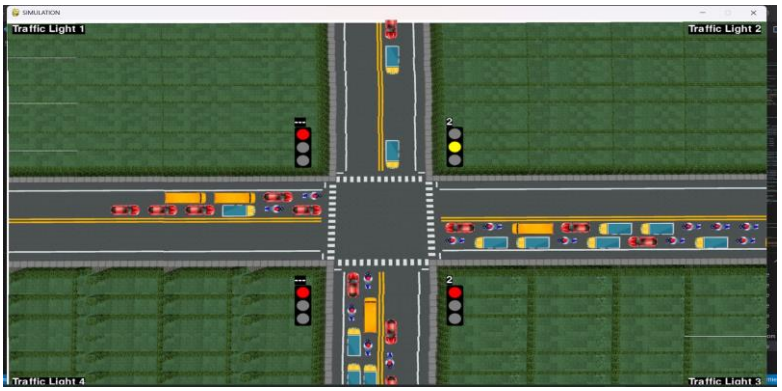

Fig 6.13. Traffic light 3 is going to get activated after 3 seconds

After each traffic signal duration ends the system prompts input for continuous operation

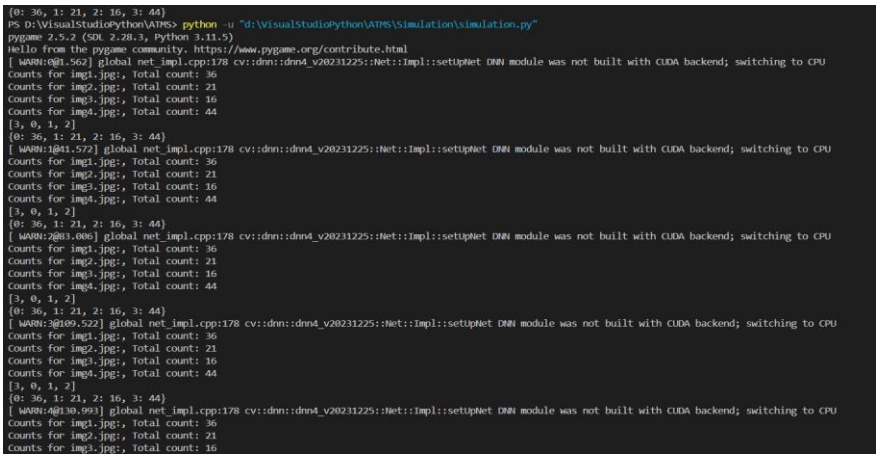

Fig 6.14. Image processing result

CHAPTER 7

CONCLUSION

The implementation of an Adaptive Traffic Management System represents a significant leap forward in addressing the inefficiencies of traditional static traffic light systems. By leveraging image processing technology, this system offers real-time monitoring, congestion reduction, and adaptive signal adjustments, leading to a substantial improvement in overall traffic flow.

Moreover, the system's ability to remotely access data for analysis and decision-making ensures efficient management of traffic even during peak hours. As the project progresses, it can be expanded to include features such as identifying vehicles violating traffic rules, detecting accidents or breakdowns, synchronizing traffic signals across multiple intersections, and prioritizing emergency vehicles.

Continued technological advancements will undoubtedly shape the future of urban transportation, and adaptive traffic management systems represent a significant step in this direction. By embracing innovation and embracing these advancements, cities can create safer, more efficient, and more sustainable transportation networks for the future.

REFERENCES

[1] Khushi, "Smart Control of Traffic Light System using Image Processing," 2017 International Conference on Current Trends in Computer, Electrical, Electronics and Communication (CTCEEC), Mysore, 2017, pp. 99-103, doi: 10.1109/CTCEEC.2017.8454966.

[2] M. M. Gandhi, D. S. Solanki, R. S. Daptardar and N. S. Baloorkar, "Smart Control of Traffic Light Using Artificial Intelligence," 2020 5th IEEE International Conference on Recent Advances and Innovations in Engineering (ICRAIE), Jaipur, India, 2020, pp. 1-6, doi: 10.1109/ICRAIE51050.2020.9358334.

[3] Chidhambaram, S & Mukilan, R & Shivabalaji, R & Vigneshwaran, K & Praba R, Dhivya & Student,. (2021). Smart Traffic Light Controller Using Image Processing. 9. 2179.

ANNEXURE

VEHICLE DETECTION:

vehicle_count.py

```
import cv2
import collections
import numpy as np
import os

class VehicleDetection:
    def __init__(self):
        self.modelConfiguration = 'yolov3-320.cfg'
        self.modelWeights = 'yolov3-320.weights'
        self.confThreshold = 0.2
        self.nmsThreshold = 0.2
        self.net = cv2.dnn.readNetFromDarknet(self.modelConfiguration, self.modelWeights)
        self.net.setPreferableBackend(cv2.dnn.DNN_BACKEND_CUDA)
        self.net.setPreferableTarget(cv2.dnn.DNN_TARGET_CUDA)
        self.classesFile = "coco.names"
        self.classNames = open(self.classesFile).read().strip().split('\n')
        self.required_class_index = [2, 3, 5, 7]

    def find_center(self, x, y, w, h):
        cx = x + w // 2
        cy = y + h // 2
        return cx, cy

    def process_image(self, image_path):
        img = cv2.imread(image_path)
        blob = cv2.dnn.blobFromImage(img, 1 / 255, (320, 320), [0, 0, 0], 1, crop=False)
        self.net.setInput(blob)
        layerNames = self.net.getLayerNames()
        outputNames = [layerNames[i - 1] for i in self.net.getUnconnectedOutLayers()]
        outputs = self.net.forward(outputNames)

        boxes = []
        confidences = []
        classIDs = []

        for output in outputs:
            for detection in output:
                scores = detection[5:]
                classID = np.argmax(scores)
                confidence = scores[classID]
```

```
            if confidence > self.confThreshold and classID in self.required_class_index:
                centerX, centerY, width, height = (detection[0:4] * np.array([img.shape[1],
img.shape[0], img.shape[1], img.shape[0]])).astype('int')
                x, y = int(centerX - width / 2), int(centerY - height / 2)
                boxes.append([x, y, int(width), int(height)])
                confidences.append(float(confidence))
                classIDs.append(classID)

        indices = cv2.dnn.NMSBoxes(boxes, confidences, self.confThreshold, self.nmsThreshold)


        vehicle_counts = collections.Counter()

        for i in indices.flatten():
            x, y, w, h = boxes[i]
            class_name = self.classNames[classIDs[i]]
            vehicle_counts[class_name] += 1

            # Draw the count label on the image
            cv2.putText(img, f"{class_name}: {vehicle_counts[class_name]}", (x, y - 5),
cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 1)

        # Save the image with count labels
        output_folder = 'processed_images_with_labels/'
        os.makedirs(output_folder, exist_ok=True)
        output_path = os.path.join(output_folder, os.path.basename(image_path))
        cv2.imwrite(output_path, img)

        return vehicle_counts

    def process_all_images(self):
        images_folder = 'images/'
        count_dict = {}

        for image_file in os.listdir(images_folder):
            if image_file.endswith('.jpg') or image_file.endswith('.png'):
                image_path = os.path.join(images_folder, image_file)
                vehicle_counts = self.process_image(image_path)
                total_count = sum(vehicle_counts.values())
                count_dict[image_file] = total_count
                print(f"Counts for {image_file}:, Total count: {total_count}")

        return count_dict

"""if __name__ == '__main__':
    detector = VehicleDetection()
    detector.process_all_images()"""
```

SIMULATION:

simulation.py

```
import random
import time
import threading
import pygame
import sys

from Vehicle_Detection.vehicle_count import VehicleDetection

# Default values of signal timers
defaultGreen = {0:20, 1:20, 2:20, 3:20}
defaultRed = 150
defaultYellow = 5
count_dict={}
count_dictKeys=[]
orgPos={}
signal_order=[]
currentGreen=0
nextGreen=0
currentYellow = 0
signals = []
current_signal_index=0
temp=-1
index=[]
sorted_values=[]
order_list_executed = False  # Flag to indicate if order_list has been executed

def order_list():
    global currentGreen, currentYellow,
nextGreen,signal_order,current_signal_index,temp,order_list_executed,index,values_list
    temp=(temp+1)%4
    detector = VehicleDetection()
    if temp==0:
        index=[]
    count_dict = detector.process_all_images()
    count_dictKeys=list(count_dict.values()) #extract keys of count_dict
    #assign original position
    for i in range(len(count_dictKeys)):
        orgPos[count_dictKeys[i]]=i
    # Convert dictionary values into a list
    values_list = list(count_dict.values())
    s=sorted(values_list,reverse=True)
    if temp!=0:
        t=[values_list[i] for i in range(len(values_list)) if i not in index]
    else:
        t=values_list
    sorted_values= sorted(t, reverse=True)
```

```
    signal_order = [orgPos[element] for element in sorted_values]
    signal_order=index+signal_order
    # Find the maximum value
    max_value = max(t)
    # Find the index of the maximum value in the list
    max_index = values_list.index(max_value)
    index.append(max_index)
    print(signal_order)

    for i in range(temp,len(signal_order)):
            defaultGreen.update({signal_order[i]:((s[i]*5)//5)})
    print(defaultGreen)

    # Indicates which signal is green currently
    currentGreen = signal_order[current_signal_index]
    nextGreen = signal_order[(current_signal_index + 1) % len(signal_order)]    # Indicates which
signal will turn green next
    currentYellow = 0   # Indicates whether yellow signal is on or off

speeds = {'car':2.25, 'bus':1.8, 'truck':1.8, 'bike':2.5}  # average speeds of vehicles

# Coordinates of vehicles' start
x = {'right':[0,0,0], 'down':[755,727,697], 'left':[1400,1400,1400], 'up':[602,627,657]}
y = {'right':[348,370,398], 'down':[0,0,0], 'left':[498,466,436], 'up':[800,800,800]}

vehicles = {'right': {0:[], 1:[], 2:[], 'crossed':0}, 'down': {0:[], 1:[], 2:[], 'crossed':0}, 'left': {0:[], 1:[],
2:[], 'crossed':0}, 'up': {0:[], 1:[], 2:[], 'crossed':0}}
vehicleTypes = {0:'car', 1:'bus', 2:'truck', 3:'bike'}
directionNumbers = {0:'right', 1:'down', 2:'left', 3:'up'}

# Coordinates of signal image, timer, and vehicle count
signalCoods = [(530,230),(810,230),(810,570),(530,570)]
signalTimerCoods = [(530,210),(810,210),(810,550),(530,550)]

# Coordinates of stop lines
stopLines = {'right': 590, 'down': 330, 'left': 800, 'up': 535}
defaultStop = {'right': 580, 'down': 320, 'left': 810, 'up': 545}
# stops = {'right': [580,580,580], 'down': [320,320,320], 'left': [810,810,810], 'up': [545,545,545]}

# Gap between vehicles
stoppingGap = 15    # stopping gap
movingGap = 15   # moving gap

pygame.init()
simulation = pygame.sprite.Group()

class TrafficSignal:
    def __init__(self, red, yellow, green):
        self.red = red
        self.yellow = yellow
```

```
        self.green = green
        self.signalText = ""

class Vehicle(pygame.sprite.Sprite):
    def __init__(self, lane, vehicleClass, direction_number, direction):
        pygame.sprite.Sprite.__init__(self)
        self.lane = lane
        self.vehicleClass = vehicleClass
        self.speed = speeds[vehicleClass]
        self.direction_number = direction_number
        self.direction = direction
        self.x = x[direction][lane]
        self.y = y[direction][lane]
        self.crossed = 0
        vehicles[direction][lane].append(self)
        self.index = len(vehicles[direction][lane]) - 1
        path = "Simulation/images/" + direction + "/" + vehicleClass + ".png"
        self.image = pygame.image.load(path)

        if(len(vehicles[direction][lane])>1 and vehicles[direction][lane][self.index-1].crossed==0):    #
if more than 1 vehicle in the lane of vehicle before it has crossed stop line
            if(direction=='right'):
                self.stop = vehicles[direction][lane][self.index-1].stop -
vehicles[direction][lane][self.index-1].image.get_rect().width - stoppingGap         # setting stop
coordinate as: stop coordinate of next vehicle - width of next vehicle - gap
            elif(direction=='left'):
                self.stop = vehicles[direction][lane][self.index-1].stop +
vehicles[direction][lane][self.index-1].image.get_rect().width + stoppingGap
            elif(direction=='down'):
                self.stop = vehicles[direction][lane][self.index-1].stop -
vehicles[direction][lane][self.index-1].image.get_rect().height - stoppingGap
            elif(direction=='up'):
                self.stop = vehicles[direction][lane][self.index-1].stop +
vehicles[direction][lane][self.index-1].image.get_rect().height + stoppingGap
        else:
            self.stop = defaultStop[direction]

        # Set new starting and stopping coordinate
        if(direction=='right'):
            temp = self.image.get_rect().width + stoppingGap
            x[direction][lane] -= temp
        elif(direction=='left'):
            temp = self.image.get_rect().width + stoppingGap
            x[direction][lane] += temp
        elif(direction=='down'):
            temp = self.image.get_rect().height + stoppingGap
            y[direction][lane] -= temp
        elif(direction=='up'):
            temp = self.image.get_rect().height + stoppingGap
            y[direction][lane] += temp
```

```
        simulation.add(self)

    def render(self, screen):
        screen.blit(self.image, (self.x, self.y))

    def move(self):
        if(self.direction=='right'):
            if(self.crossed==0 and self.x+self.image.get_rect().width>stopLines[self.direction]):   # if
the image has crossed stop line now
                self.crossed = 1
            if((self.x+self.image.get_rect().width<=self.stop or self.crossed == 1 or (currentGreen==0
and currentYellow==0)) and (self.index==0 or
self.x+self.image.get_rect().width<(vehicles[self.direction][self.lane][self.index-1].x -
movingGap))):
            # (if the image has not reached its stop coordinate or has crossed stop line or has green
signal) and (it is either the first vehicle in that lane or it is has enough gap to the next vehicle in that
lane)
                self.x += self.speed  # move the vehicle
        elif(self.direction=='down'):
            if(self.crossed==0 and self.y+self.image.get_rect().height>stopLines[self.direction]):
                self.crossed = 1
            if((self.y+self.image.get_rect().height<=self.stop or self.crossed == 1 or (currentGreen==1
and currentYellow==0)) and (self.index==0 or
self.y+self.image.get_rect().height<(vehicles[self.direction][self.lane][self.index-1].y -
movingGap))):
                self.y += self.speed
        elif(self.direction=='left'):
            if(self.crossed==0 and self.x<stopLines[self.direction]):
                self.crossed = 1
            if((self.x>=self.stop or self.crossed == 1 or (currentGreen==2 and currentYellow==0)) and
(self.index==0 or self.x>(vehicles[self.direction][self.lane][self.index-1].x +
vehicles[self.direction][self.lane][self.index-1].image.get_rect().width + movingGap))):
                self.x -= self.speed
        elif(self.direction=='up'):
            if(self.crossed==0 and self.y<stopLines[self.direction]):
                self.crossed = 1
            if((self.y>=self.stop or self.crossed == 1 or (currentGreen==3 and currentYellow==0)) and
(self.index==0 or self.y>(vehicles[self.direction][self.lane][self.index-1].y +
vehicles[self.direction][self.lane][self.index-1].image.get_rect().height + movingGap))):
                self.y -= self.speed

# Initialization of signals with default values
def initialize():
    a=signal_order[0]
    b=a=signal_order[1]
    ts1 = TrafficSignal(defaultRed, defaultYellow, defaultGreen[0])
    signals.append(ts1)
    ts2 = TrafficSignal(defaultRed, defaultYellow, defaultGreen[1])
    signals.append(ts2)
    ts3 = TrafficSignal(defaultRed, defaultYellow, defaultGreen[2])
```

```
    signals.append(ts3)
    ts4 = TrafficSignal(defaultRed, defaultYellow, defaultGreen[3])
    signals.append(ts4)

    signals[a]=TrafficSignal(0, defaultYellow, defaultGreen[a])
    tsa = signals[a]
    signals[0]=TrafficSignal(tsa.red+tsa.yellow+tsa.green, defaultYellow, defaultGreen[0])
    repeat()

def repeat():
    global currentGreen, currentYellow, nextGreen,current_signal_index
    while(signals[currentGreen].green>0):   # while the timer of current green signal is not zero
        updateValues()
        time.sleep(1)
        if (signals[currentGreen].green==15):
            # Start a new thread to execute order_list() while the timer continues
            order_thread = threading.Thread(target=order_list)
            order_thread.start()
    currentYellow = 1   # set yellow signal on
    # reset stop coordinates of lanes and vehicles
    for i in range(0,3):
        for vehicle in vehicles[directionNumbers[currentGreen]][i]:
            vehicle.stop = defaultStop[directionNumbers[currentGreen]]
    while(signals[currentGreen].yellow>0):  # while the timer of current yellow signal is not zero
        updateValues()
        time.sleep(1)
    currentYellow = 0   # set yellow signal off

     # reset all signal times of current signal to default times
    signals[currentGreen].green = defaultGreen[currentGreen]
    signals[currentGreen].yellow = defaultYellow
    signals[currentGreen].red = defaultRed

    currentGreen = nextGreen # set next signal as green signal
    current_signal_index =(current_signal_index+1)%4
    nextGreen = signal_order[(current_signal_index + 1) % len(signal_order)]    # set next green
signal
    signals[nextGreen].red = signals[currentGreen].yellow+signals[currentGreen].green    # set the
red time of next to next signal as (yellow time + green time) of next signal
    repeat()

# Update values of the signal timers after every second
def updateValues():
    for i in signal_order:
        if(i==currentGreen):
            if(currentYellow==0):
                signals[i].green-=1
            else:
                signals[i].yellow-=1
        else:
```

```
            signals[i].red-=1

# Generating vehicles in the simulation
def generateVehicles():
    while(True):
        vehicle_type = random.randint(0,3)
        lane_number = random.randint(1,2)
        temp = random.randint(0,99)
        direction_number = 0
        dist = [25,50,75,100]
        if(temp<dist[0]):
            direction_number = 0
        elif(temp<dist[1]):
            direction_number = 1
        elif(temp<dist[2]):
            direction_number = 2
        elif(temp<dist[3]):
            direction_number = 3
        Vehicle(lane_number, vehicleTypes[vehicle_type], direction_number,
directionNumbers[direction_number])
        time.sleep(1)

class Main:
    order_list()
    thread1 = threading.Thread(name="initialization",target=initialize, args=())    # initialization
    thread1.daemon = True
    thread1.start()

    # Colours
    black = (0, 0, 0)
    white = (255, 255, 255)

    # Screensize
    screenWidth = 1400
    screenHeight = 800
    screenSize = (screenWidth, screenHeight)

    # Setting background image i.e. image of intersection
    background = pygame.image.load('Simulation/images/intersection.png')

    screen = pygame.display.set_mode(screenSize)
    pygame.display.set_caption("SIMULATION")

    # Loading signal images and font
    redSignal = pygame.image.load('Simulation/images/signals/red.png')
    yellowSignal = pygame.image.load('Simulation/images/signals/yellow.png')
    greenSignal = pygame.image.load('Simulation/images/signals/green.png')
    font = pygame.font.Font(None, 30)

    thread2 = threading.Thread(name="generateVehicles",target=generateVehicles, args=())    #
```

```
Generating vehicles
    thread2.daemon = True
    thread2.start()

    while True:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                sys.exit()

        screen.blit(background,(0,0))   # display background in simulation

        # Display labels in the four corners
        top_left_label = font.render("Traffic Light 1", True, white, black)
        screen.blit(top_left_label, (10, 10))

        top_right_label = font.render("Traffic Light 2", True, white, black)
        screen.blit(top_right_label, (screenWidth - top_right_label.get_width() - 10, 10))

        bottom_left_label = font.render("Traffic Light 4", True, white, black)
        screen.blit(bottom_left_label, (10, screenHeight - bottom_left_label.get_height() - 10))

        bottom_right_label = font.render("Traffic Light 3", True, white, black)
        screen.blit(bottom_right_label, (screenWidth - bottom_right_label.get_width() - 10,
screenHeight - bottom_right_label.get_height() - 10))


        for i in signal_order:  # display signal and set timer according to current status: green, yello, or
red
            if(i==currentGreen):
                if(currentYellow==1):
                    signals[i].signalText = signals[i].yellow
                    screen.blit(yellowSignal, signalCoods[i])
                else:
                    signals[i].signalText = signals[i].green
                    screen.blit(greenSignal, signalCoods[i])
            else:
                if(signals[i].red<=10):
                    signals[i].signalText = signals[i].red
                else:
                    signals[i].signalText = "---"
                screen.blit(redSignal, signalCoods[i])
        signalTexts = ["","","",""]

        # display signal timer
        for i in signal_order:
            signalTexts[i] = font.render(str(signals[i].signalText), True, white, black)
            screen.blit(signalTexts[i],signalTimerCoods[i])

        # display the vehicles
        for vehicle in simulation:
```

```
            screen.blit(vehicle.image, [vehicle.x, vehicle.y])
            vehicle.move()
        pygame.display.update()


Main()
```