Smart Internz

PARKING GUIDANCE SYSTEM

A project report submitted by

Madhan Kumar S (20BEC1112)
Abhishek Sebastian (20BEC1118)
Pragna R (20BLC1104)

Under the guidance of
Sonu

Smart Internz

TABLE OF CONTENTS

1. INTRODUCTION

1.1 **OVERVIEW**

The surge in vehicle ownership has led to congested parking lots, resulting in frustrating experiences for drivers and diminished operational efficiency for businesses. When parking spaces are utilized inefficiently, it can lead to a range of problems and challenges. As the number of vehicles continues to rise in urban centers, the need for efficient parking management becomes paramount. Traditional parking practices are proving inadequate in meeting the demands of the growing number of cars entering high demand areas such as airports, malls and hotels. This work presents a Parking Guidance System (PGS) which monitors the vehicles moment and allots the best parking slot for the entering vehicle the parking lot by analyzing the vehicle characteristics. By leveraging technology and real-time data, PGS offers an effective solution to alleviate parking-related challenges, enhance customer experiences, optimize space utilization, and promote sustainable urban mobility.

1.2 PURPOSE

The rapid increase in vehicle ownership and urbanization has led to a significant rise in the number of cars entering high-demand areas such as airports, malls, hotels, and other crowded locations. As a result, the availability and efficient management of parking spaces have become critical challenges for both drivers and businesses operating in these areas. Traditional parking practices are struggling to cope with the growing demand, leading to congestion, frustration, and decreased operational efficiency. To address these pressing issues, the implementation of a Parking Guidance System (PGS) has emerged as a necessary solution to optimize parking utilization and enhance the overall parking experience.

High-demand areas face unique parking challenges due to the sheer volume of vehicles and the limited availability of parking spaces. Drivers often spend valuable time searching for an open spot, resulting in traffic congestion, delays, and an overall negative impact on the user experience. Additionally, businesses in these areas face the dual challenge of maximizing revenue from limited parking resources while ensuring a seamless and convenient parking experience for their customers.

A Parking Guidance System (PGS) offers an innovative and technology-driven approach to tackle these challenges effectively.

In culmination, this work contributes to the following.

- Designed a novel system capable of detecting the car type and the number plate. This system can classify the cars based on their size and read the number plate using deep learning algorithm (YOLOv8).
- Designed a smart parking allotment system that can allocate appropriate parking slot based on the identified car characteristics.

PGS provides a practical approach to address parking-related issues, improve customer satisfaction, promote space utilization, and support sustainable urban transportation.

2. LITERATURE SURVEY

2.1 EXISTING SYSTEMS

Parking Guidance Systems (PGS) have gained popularity over the years, and various systems have been developed to cater to the diverse needs of parking facilities. This section will discuss the indigenous systems present in the market, such as single-space sensors, occupancy sensors, and license plate recognition systems. The systems' features, advantages, and limitations will be analyzed to provide an understanding of their suitability for different types of parking facilities.

SI. No	Name Of the System	Link
1	Parking Guidance Systems, LLC	https://parkingguidancesystems.com/my-product/indoor/

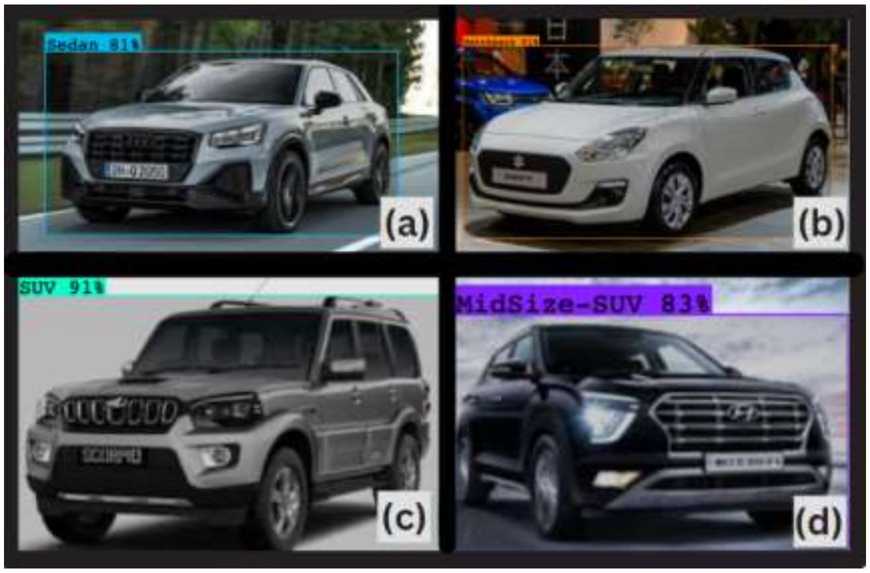
SI. No	Name Of the System	Link
2	Auto pass India	https://www.autopassindia.com/pdf/auto-pass-india-brochure-2017.pdf

SI. No	Name Of the System	Link
3	Houston systems	https://www.houstonsystem.com/solutions-2/parking-guidance-solutions/intelligent-guidance-system/

2.2 PROPOSED SYSTEM

Car Type Classification:

We employ the YOLOv8 deep learning algorithm, known for its robust object detection capabilities, to classify car types based on their size. The system is trained on a diverse dataset containing images of various car types and sizes. We fine-tune the pre-trained YOLOv8 model on our dataset and optimize the hyperparameters to achieve accurate and reliable car type classification.



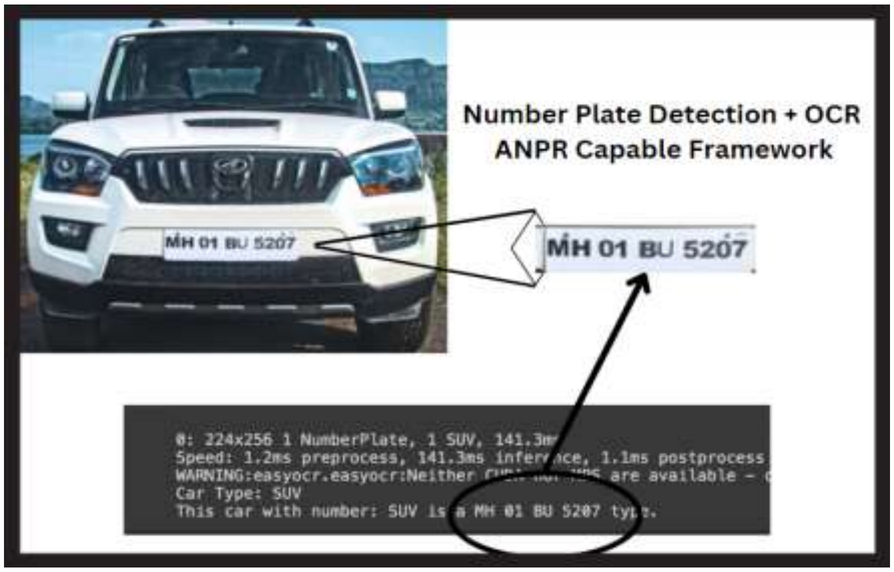
License Plate Recognition:

The proposed system incorporates the YOLOv8 architecture for license plate recognition. The model is trained using a substantial dataset containing annotated license plate images. The training process involves several stages, including data preprocessing to optimize the model's performance. The resulting model demonstrates a high level of accuracy in recognizing license plates, regardless of variations in lighting conditions and perspectives.

To detect the license plate, the YOLOv8 model is employed. Once the license plate is identified, the system crops the image to isolate the license plate region for further processing. Easy OCR, a widely-used optical character recognition (OCR) library, is then utilized to extract the number from the cropped license plate image.

Easy OCR provides efficient and accurate text extraction capabilities, allowing the system to retrieve the alphanumeric characters present on the license plate. The extracted number can be further processed or modified as needed to fit specific requirements, such as formatting or validation.

By combining the YOLOv8 model for license plate detection and Easy OCR for text extraction, the proposed system achieves reliable and accurate license plate recognition. This integrated approach enables the system to effectively read license plate numbers from a variety of images, enabling subsequent actions or analysis based on the extracted information.



```
0: 224x256 1 NumberPlate, 1 SUV, 141.3ms
Speed: 1.2ms preprocess, 141.3ms inference, 1.1ms postprocess
WARNING:easyocr.easyocr:Neither CUDA nor MPS are available - 
Car Type: SUV
This car with number: SUV is a MH 01 BU 5207 type.
```

Parking Slot Allocation:

The parking slot allocation algorithm is designed to assign parking spaces to cars based on their type and number, while adhering to certain predefined conditions. These conditions ensure that specific rules are followed during the allocation process. Here is an explanation of the algorithm in a more generic language:

Initialize variables:

- Keep track of the available parking spaces for each car type (SUV, Sedan, Hatchback, Mid Size SUV).
- Maintain counters for each car type to monitor the allocated spaces.
- Create a list to store the allocated parking spaces.

Check the car type:

- If the car type is an SUV, verify if the number of allocated SUV spaces is below the desired threshold (40% of the total parking lots on the floor).
- If there is room for another SUV, assign an SUV space.
- If the threshold is exceeded, proceed to the next step.

Check the car type:

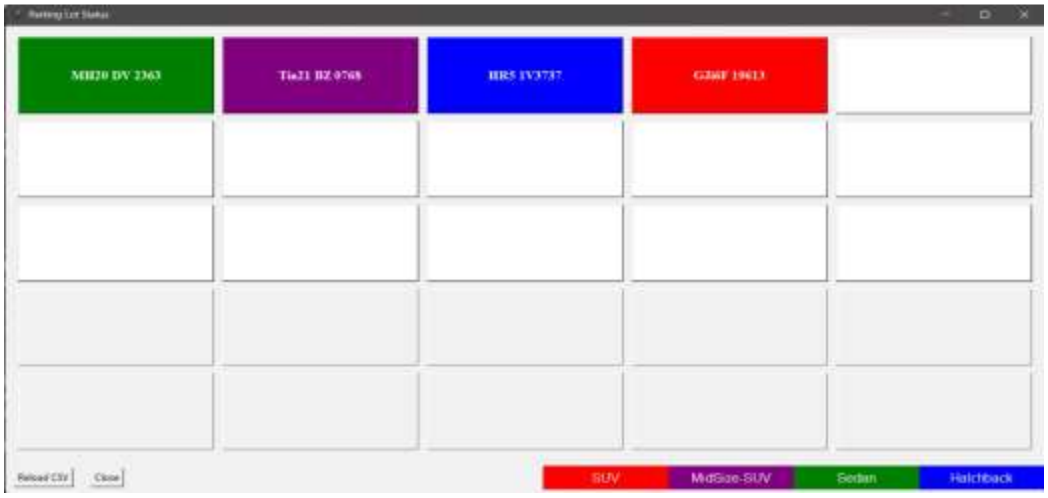
- If the car type is a Sedan or Hatchback, check if there are at least two available spaces for Sedans or Hatchbacks.
- If there are two spaces available, assign a parking space for the Sedan or Hatchback.
- If there are not enough spaces, move to the next step.
- If the car type is a Mid Size SUV, check if there are at least two available spaces for Mid Size SUVs, Sedans, or Hatchbacks.
- If there are two spaces available, assign a parking space for the Mid Size SUV.
- If there are not enough spaces, proceed to the next step.

Assign a parking space based on the remaining available spaces:

- If there are still SUV spaces available, assign an SUV space.
- If there are no SUV spaces available, but Sedan or Hatchback spaces are available, assign a parking space for a Sedan or Hatchback.
- If there are no SUV, Sedan, or Hatchback spaces available, assign a parking space for a Mid Size SUV.
- Update the counters and the list of allocated parking spaces.

Repeat the above steps for each incoming car.

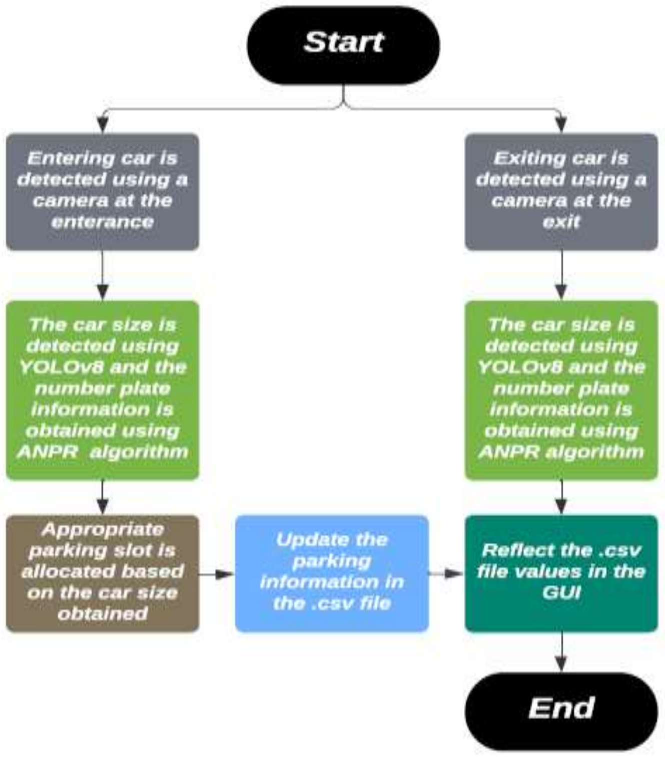
By following these steps, the algorithm ensures that parking slots are allocated according to the predefined conditions. These conditions aim to prevent the proximity of two SUV type cars, maintain the desired percentage of SUV spaces, allow Sedans and Hatchbacks to be placed closely, and permit a Mid Size SUV to be located between two Mid Size SUVs, Sedans, or Hatchbacks.



It's important to note that this algorithm assumes the availability of a predefined set of parking lots and does not account for real-time occupancy or dynamic allocation based on changing conditions.

3. THEORITICAL ANALYSIS

3.1 BLOCK DIAGRAM


Start
Entering car is detected using a camera at the enterance
Exiting car is detected using a camera at the exit
The car size is detected using YOLOv8 and the number plate information is obtained using ANPR algorithm
The car size is detected using YOLOv8 and the number plate information is obtained using ANPR algorithm
Appropriate parking slot is allocated based on the car size obtained
Update the parking information in the .csv file
Reflect the .csv file values in the GUI
End

3.2 HARDWARE/ SOFTWARE DESIGNING

YOLOv8:

YOLO (You Only Look Once), a popular object detection and image segmentation model, was initially introduced in 2015 by Joseph Redmon and Ali Farhadi at the University of Washington. Its innovative approach revolutionized real-time object detection by adopting a single-pass architecture that simultaneously predicts object bounding boxes and class probabilities. This resulted in significantly faster inference times compared to traditional object detection models.

Since its inception, YOLO has undergone several iterations, with each version introducing improvements and advancements to enhance its performance and capabilities. YOLOv2, released in 2016, incorporated batch normalization, anchor boxes, and dimension clusters, leading to improved accuracy and robustness. YOLOv3, launched in 2018, further refined the model with a more efficient backbone network, multiple anchors, and spatial pyramid pooling, achieving even better object detection results.

Building upon the success of previous versions, YOLOv4 was released in 2020, introducing cutting-edge techniques such as Mosaic data augmentation, an anchor-free detection head, and a new loss function. YOLOv5, released shortly after, focused on improving performance through hyperparameter optimization, integrated experiment tracking, and automatic export to popular export formats. YOLOv6, open-sourced by Meituan in 2022, found applications in autonomous delivery robots, while YOLOv7 expanded its capabilities to include pose estimation on the COCO keypoints dataset.

Now, Ultralytics presents YOLOv8, the latest iteration of the YOLO model. Combining the advancements and insights gained from previous versions, YOLOv8 introduces new features and improvements, solidifying its position as a state-of-the-art (SOTA) model for object detection and image segmentation. Its streamlined architecture, exceptional speed, and high accuracy make it ideal for integration into various software systems, including our Parking Guidance System (PGS).

By integrating YOLOv8 into our PGS software, we harness its powerful object detection capabilities to accurately identify and classify different car types in real-time. This integration empowers our PGS to effectively allocate parking slots based on car characteristics, ensuring optimized utilization and enhanced parking management. Leveraging YOLOv8's speed and accuracy, our PGS can provide real-time information to drivers, guiding them to available parking spaces and improving the overall parking experience.

EasyOCR:

Easy OCR is a vital component in integrating the Parking Guidance System (PGS) with the license plate recognition module. It utilizes advanced machine learning algorithms for accurate text extraction from license plate images, regardless of font, size, or style. With robust image processing techniques, including noise reduction and contrast enhancement, Easy OCR enhances the quality of license plate images, ensuring improved OCR accuracy even in challenging conditions. Its multilingual support enables accurate recognition of license plate numbers in different languages and character sets. Easy OCR's real-time processing capabilities enable swift extraction of license plate numbers, facilitating efficient handling of vehicles in parking operations. Furthermore, Easy OCR offers seamless integration with the PGS, allowing smooth communication and data exchange between the license plate recognition module and other system components.

Camera:

The camera plays a crucial role in the integration of the Parking Guidance System (PGS) with YOLOv8. The camera serves as the primary input source, capturing live feed from the parking area and providing visual data for object detection and classification. The capabilities of the camera are essential for the successful implementation of the PGS.

The camera used in the system should possess key capabilities to ensure accurate and reliable object detection. These capabilities include high-resolution imaging, enabling clear and detailed visuals of the parking area. Additionally, the camera should have good low-light performance to ensure visibility in various lighting conditions, such as dimly lit or outdoor environments.

4. EXPERIMENTAL INVESTIGATIONS

The experimental investigations conducted for this project involve the evaluation and validation of the proposed Parking Guidance System (PGS) in real-world scenarios. The following are some key areas of experimental investigation:

- Car Detection and Classification Performance: The performance of the YOLOv8 model in accurately detecting and classifying different types of vehicles is evaluated. This involves collecting a diverse dataset of vehicle images and testing the model's ability to correctly identify the car type, such as SUVs, sedans, hatchbacks, and mid-size SUVs. The evaluation metrics may include precision, recall, and accuracy.

- License Plate Recognition Accuracy: The Easy OCR component is assessed for its accuracy in recognizing and extracting license plate numbers. A dataset of license plate images with varying conditions, such as different lighting, angles, and occlusions, is used to evaluate the OCR algorithm's performance. The evaluation metrics may include character recognition accuracy and overall license plate recognition rate.

- Real-Time Processing and System Responsiveness: The PGS's real-time processing capabilities are tested to ensure its responsiveness in detecting and allocating parking spaces as vehicles enter or exit. The system's ability to handle multiple concurrent requests and provide timely updates on parking availability is evaluated. Metrics such as response time and throughput are measured to assess system performance.

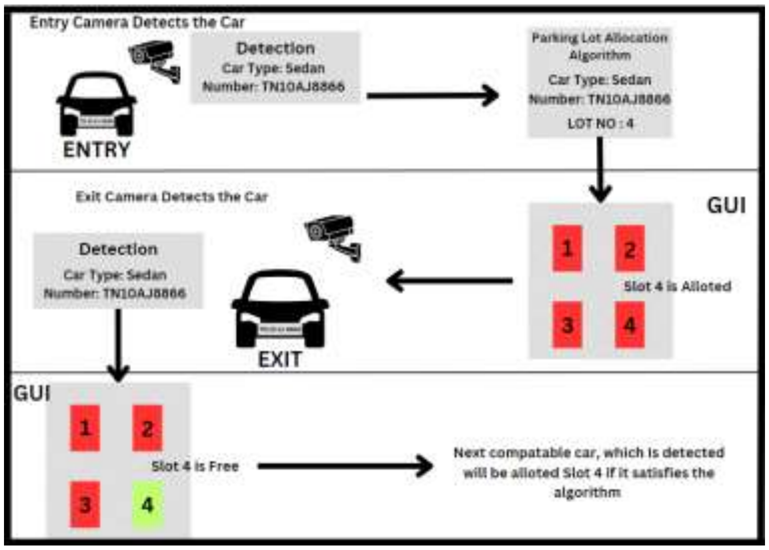
- Parking Slot Allocation Algorithm Validation: The proposed parking slot allocation algorithm is validated to ensure that it meets the predefined conditions and effectively allocates parking spaces based on vehicle characteristics. The algorithm is tested with various scenarios and datasets to verify its performance in maintaining the desired distribution of car types and optimizing parking space utilization.

- User Experience Evaluation: The user experience of the PGS is assessed through surveys, feedback collection, and user testing. Participants are asked to interact with the system, locate parking spaces based on the guidance provided, and provide feedback on the system's effectiveness, ease of use, and overall satisfaction.

- Integration and Compatibility Testing: The integration of the PGS with existing parking infrastructure, such as cameras, sensors, and payment systems, is tested to ensure seamless communication and data exchange. Compatibility with different hardware platforms, operating systems, and APIs is evaluated to assess the system's adaptability and scalability.

These experimental investigations aim to validate the performance, accuracy, responsiveness, and user satisfaction of the proposed PGS. The results obtained from these experiments provide valuable insights into the system's effectiveness, identify potential areas for improvement, and demonstrate the feasibility and practicality of deploying the PGS in real-world parking scenarios.

5. FLOW CHART



The proposed system follows a detailed flow for efficient parking slot allocation and management. The flow is as follows:

Car Entry:

- When a car enters the parking area, the system's cameras detect the car and capture its type and number plate information.
- The captured information is then processed using the YOLOv8-based deep learning algorithm to identify the car type accurately and extract the license plate number.

Slot Allocation:

- Based on the identified car characteristics, the system's parking slot allocation algorithm comes into play.

- The algorithm considers various factors, such as the car type, predefined conditions, and available parking slots, to allocate an appropriate parking space for the incoming car.
- It ensures that the allocated slot meets the specific conditions, such as preventing the proximity of two SUVs and maintaining a balanced distribution of car types within the parking area.
- Once the algorithm determines the suitable parking slot, it assigns the slot to the car.

Car Exit:

- When the car exits the parking area, the algorithm marks the allocated slot as vacant and available for the next vehicle.
- The system waits for the next compatible car to enter the parking area, triggering the car detection and slot allocation process again.

Continuous Monitoring:

- Throughout the process, the system continuously monitors the parking area using cameras to detect any changes in occupancy and update the availability of parking slots.
- The algorithm dynamically adjusts the slot allocation based on the real-time status of the parking area.

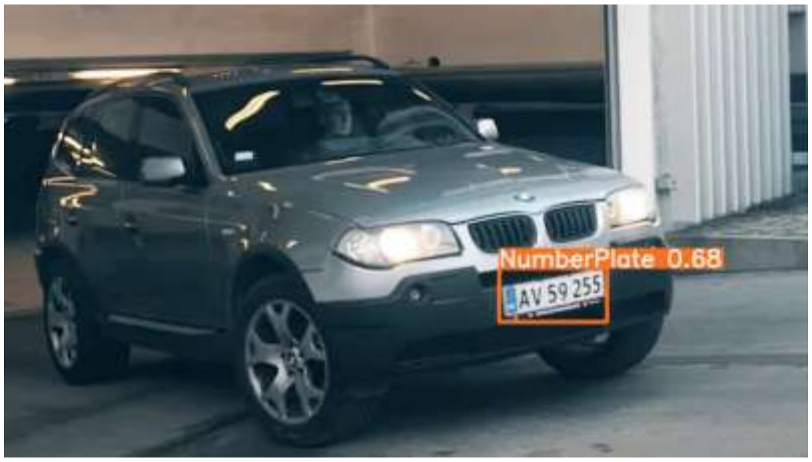
By following this detailed flow, the proposed system ensures efficient parking slot allocation and management. It seamlessly handles car entry, detects car characteristics, allocates appropriate slots based on predefined conditions, releases slots upon car exit, and continuously monitors the parking area for real-time updates. This comprehensive approach optimizes parking space utilization, reduces congestion, and enhances the overall parking experience for users.

6. RESULT

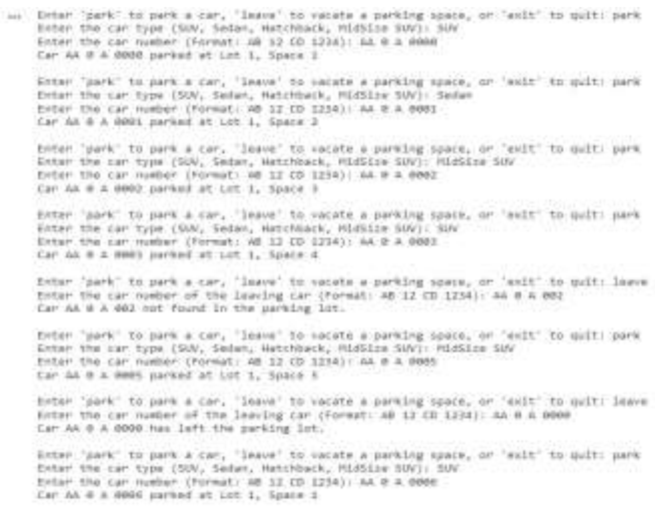
The final outputs of the proposed Parking Guidance System (PGS) project include:

Car Type Classification: The system accurately classifies vehicles entering the parking area into different types, such as SUVs, sedans, hatchbacks, and mid-size SUVs. The output includes the classification label assigned to each vehicle based on its characteristics.

License Plate Recognition: The PGS successfully recognizes and extracts license plate numbers from the captured license plate images. The output includes the extracted alphanumeric characters, representing the license plate number of each vehicle.

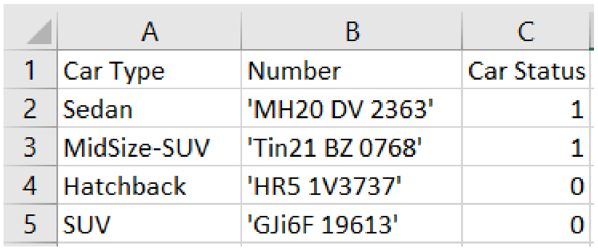


Parking Slot Allocation: The PGS algorithm allocates appropriate parking slots to incoming vehicles based on their detected type and predefined conditions. The output includes the assigned parking slot number for each vehicle, ensuring efficient and optimized parking space utilization.



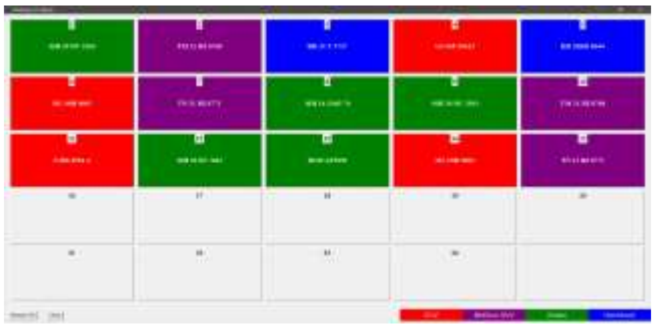
```
Enter 'park' to park a car, 'leave' to vacate a parking space, or 'exit' to quit: park
Enter the car type (SUV, Sedan, Hatchback, MidSize SUV): SUV
Enter the car number (Format: AB 12 CD 1234): AA 0 A 0000
Car AA 0 A 0000 parked at Lot 1, Space 1

Enter 'park' to park a car, 'leave' to vacate a parking space, or 'exit' to quit: park
Enter the car type (SUV, Sedan, Hatchback, MidSize SUV): Sedan
Enter the car number (Format: AB 12 CD 1234): AA 0 A 0001
Car AA 0 A 0001 parked at Lot 1, Space 2

Enter 'park' to park a car, 'leave' to vacate a parking space, or 'exit' to quit: park
Enter the car type (SUV, Sedan, Hatchback, MidSize SUV): MidSize SUV
Enter the car number (Format: AB 12 CD 1234): AA 0 A 0002
Car AA 0 A 0002 parked at Lot 1, Space 3

Enter 'park' to park a car, 'leave' to vacate a parking space, or 'exit' to quit: park
Enter the car type (SUV, Sedan, Hatchback, MidSize SUV): SUV
Enter the car number (Format: AB 12 CD 1234): AA 0 A 0003
Car AA 0 A 0003 parked at Lot 1, Space 4

Enter 'park' to park a car, 'leave' to vacate a parking space, or 'exit' to quit: leave
Enter the car number of the leaving car (Format: AB 12 CD 1234): AA 0 A 002
Car AA 0 A 002 not found in the parking lot.

Enter 'park' to park a car, 'leave' to vacate a parking space, or 'exit' to quit: park
Enter the car type (SUV, Sedan, Hatchback, MidSize SUV): MidSize SUV
Enter the car number (Format: AB 12 CD 1234): AA 0 A 0005
Car AA 0 A 0005 parked at Lot 1, Space 5

Enter 'park' to park a car, 'leave' to vacate a parking space, or 'exit' to quit: leave
Enter the car number of the leaving car (Format: AB 12 CD 1234): AA 0 A 0000
Car AA 0 A 0000 has left the parking lot.

Enter 'park' to park a car, 'leave' to vacate a parking space, or 'exit' to quit: park
Enter the car type (SUV, Sedan, Hatchback, MidSize SUV): SUV
Enter the car number (Format: AB 12 CD 1234): AA 0 A 0006
Car AA 0 A 0006 parked at Lot 1, Space 1
```

Real-Time Updates: The system provides real-time updates on parking availability, indicating the number of vacant slots and their corresponding locations. These updates enable drivers to make informed decisions and quickly find available parking spaces.



	A	B	C
1	Car Type	Number	Car Status
2	Sedan	'MH20 DV 2363'	1
3	MidSize-SUV	'Tin21 BZ 0768'	1
4	Hatchback	'HR5 1V3737'	0
5	SUV	'GJi6F 19613'	0

User Interface: The PGS features a user-friendly interface that displays the parking guidance information to drivers. The interface may include a map of the parking area, with clearly marked vacant and occupied slots, and directions guiding drivers to their allocated parking spaces.



The final outputs of the project demonstrate the successful implementation of the PGS, providing accurate car type classification, license plate recognition, optimized parking slot allocation, real-time updates, and a user-friendly interface. These outputs contribute to enhanced parking management, reduced congestion, improved user experience, and the potential for data-driven decision-making in parking operations.

7. ADVANTAGES AND DISADVANTAGES

Advantages of the proposed Parking Guidance System (PGS):

- Efficient Parking Space Allocation: The PGS, utilizing the car detection and classification capabilities of YOLOv8, enables accurate identification and classification of vehicles. This allows for efficient allocation of parking spaces based on the car characteristics, ensuring optimal utilization of available parking slots.

- Real-Time Updates: The integration of YOLOv8 and Easy OCR into the PGS provides real-time monitoring and updates. The system can promptly detect the entry and exit of vehicles, update parking availability information, and guide drivers to available parking spaces, reducing search time and congestion.

- Enhanced Parking Management: The PGS improves overall parking management by automating the allocation process. The algorithm ensures that predefined conditions, such as maintaining a balanced distribution of car types, are met, preventing overcrowding of specific vehicle types and optimizing parking space utilization.

- Improved User Experience: With real-time information on parking availability and guidance to vacant spots, the PGS enhances the overall user experience. Drivers can quickly locate available parking spaces, reducing frustration and saving time.

- Integration Potential: The proposed PGS can be easily integrated into existing parking infrastructures. The utilization of YOLOv8 and Easy OCR as software components allows for flexible integration, making it adaptable to different parking environments and systems.

Disadvantages of the proposed Parking Guidance System (PGS):

- Initial Setup and Investment: Implementing the PGS requires initial setup and investment in the necessary hardware, such as cameras and processing systems. Additionally, training the YOLOv8 model and preparing the dataset for Easy OCR may require significant resources and expertise.

- Dependency on Image Quality: The accuracy of the PGS relies on the quality of the captured images. Factors such as poor lighting conditions or obscured license plates may affect the performance of the system, leading to potential inaccuracies in car detection and license plate recognition.

- Maintenance and Upkeep: The PGS requires regular maintenance and upkeep to ensure the proper functioning of cameras, software components, and algorithm updates. Failure to maintain the system adequately may lead to decreased accuracy and reliability over time.

- Limited Compatibility: The PGS may face compatibility challenges when integrating with existing parking systems that use different technologies or have unique requirements. Ensuring seamless integration and interoperability may require additional efforts and adjustments.

- Privacy and Data Security Concerns: The use of cameras and license plate recognition technology raises privacy concerns. Proper measures should be in place to protect the collected data, ensuring compliance with privacy regulations and preventing unauthorized access or misuse.

It is important to consider these advantages and disadvantages when evaluating the feasibility and implementation of the proposed Parking Guidance System (PGS). Addressing potential challenges and ensuring proper maintenance and security measures can contribute to the successful deployment and operation of the system.

8. APPLICATIONS

The proposed Parking Guidance System (PGS) has a wide range of potential applications across various industries and settings. Some key applications for the PGS include:

- Urban Parking Management: The PGS can be deployed in urban areas, helping cities manage parking spaces more efficiently. By providing real-time updates on parking availability and guiding drivers to vacant spots, the PGS reduces traffic congestion, minimizes the time spent searching for parking, and enhances overall parking management.

- Commercial Parking Facilities: The PGS can be implemented in commercial parking facilities such as shopping malls, airports, and stadiums. It assists drivers in locating available parking spaces quickly and conveniently, improving customer satisfaction and optimizing parking space utilization.

- Workplace Parking: The PGS can be utilized in office complexes and corporate campuses to streamline employee parking. It ensures fair allocation of parking slots and provides real-time information on available spaces, reducing parking-related disputes and maximizing parking efficiency.

- Residential Parking: Residential areas with limited parking spaces can benefit from the PGS. By allocating parking slots based on vehicle characteristics and guiding residents to available spots, the system helps alleviate parking congestion and enhances residential parking management.

- Smart City Integration: Integrating the PGS with smart city initiatives allows for comprehensive traffic and parking management. By integrating with existing traffic management systems, public transportation networks, and mobile applications, the PGS contributes to creating a smarter and more sustainable urban environment.

- Parking Guidance for Special Needs: The PGS can incorporate features that cater to individuals with special needs, such as designated accessible parking spaces and guidance for accessible routes. This ensures equitable access to parking facilities for all users.

- Event Parking Management: During large events or festivals, the PGS can assist event organizers in efficiently managing parking spaces. It can guide attendees to available parking areas, prevent overcrowding, and optimize traffic flow around the event venue.

- Smart Transportation Systems: The PGS can be integrated with smart transportation systems, including connected vehicles and intelligent traffic signals. By providing real-time parking availability information, the system enables drivers to make informed decisions and contributes to overall traffic optimization.

These applications demonstrate the versatility and potential impact of the proposed PGS in various contexts. By improving parking management, reducing congestion, and enhancing user experience, the system contributes to more efficient and sustainable transportation systems.

9. CONCLUSIONS

The proposed Parking Guidance System (PGS) incorporating YOLOv8 for car detection and classification, as well as Easy OCR for license plate recognition, offers a promising solution for efficient parking space allocation and enhanced parking management. By leveraging advanced deep learning algorithms and optical character recognition techniques, the PGS provides real-time updates on parking availability, guides drivers to vacant spots, and optimizes parking space utilization.

The integration of YOLOv8 and Easy OCR into the PGS demonstrates the potential for intelligent parking systems that leverage computer vision and machine learning. The system's ability to accurately detect and classify vehicles, extract license plate numbers, and allocate parking slots based on predefined conditions has the potential to greatly streamline parking operations, reduce congestion, and improve the overall user experience.

10. FUTURE SCOPE

The proposed PGS lays the foundation for further advancements and enhancements in parking guidance systems. Some potential areas for future exploration and development include:

- Scalability and Adaptability: Further research can focus on scaling the PGS to handle larger parking areas with multiple entrances and exits. Additionally, exploring the adaptability of the system to different parking lot layouts and configurations can enhance its versatility.

- Integration with Smart City Infrastructure: Integrating the PGS with other smart city infrastructure, such as traffic management systems and mobile applications, can provide a comprehensive solution for efficient parking and traffic flow optimization.

- Enhanced Security and Privacy Measures: Future iterations of the PGS should prioritize robust security measures to protect sensitive data, including license plate numbers and user information. Implementing encryption techniques and complying with privacy regulations will ensure user trust and data integrity.

- Predictive Analytics and Machine Learning: Incorporating predictive analytics and machine learning algorithms into the PGS can enable proactive decision-making, such as predicting parking demand patterns, optimizing parking slot allocations based on historical data, and dynamically adjusting parking policies.

- Integration with Payment Systems: Integrating the PGS with digital payment systems can provide a seamless and convenient experience for users, enabling cashless transactions and automating payment processes.

Overall, the proposed PGS demonstrates the potential for intelligent parking guidance systems that leverage deep learning and computer vision technologies. Continued research and development in these areas can lead to more efficient, sustainable, and user-friendly parking solutions, contributing to the advancement of smart cities and urban infrastructure.

10. BIBLOGRAPHY

1. https://docs.ultralytics.com/

2. https://github.com/JaidedAI/EasyOCR

3. https://parkingguidancesystems.com/my-product/indoor/

4. https://www.autopassindia.com/pdf/auto-pass-india-brochure-2017.pdf

5. https://www.houstonsystem.com/solutions-2/parking-guidance-solutions/intelligent-guidance-system/

6. https://docs.python.org/3/library/tk.html

11.2 APPENDIX

```
from ultralytics import YOLO import cv2

import os
import easyocr

import re

def extract_text_from_image(image_path):

    # Initialize the EasyOCR reader

    reader = easyocr.Reader(['en'])

    # Read the image and extract text

    result = reader.readtext(image_path)

    # Extract all text

    extracted_text = []

    for detection in result:

        text = detection[1]

        extracted_text.append(text)

    return extracted_text

import csv

def create_csv_file():

    # Define the file path for the CSV file

    csv_file = 'vehicle_information.csv'

    # Create the CSV file

    with open(csv_file, mode='w', newline='') as file:

        writer = csv.writer(file)

        writer.writerow(['Car ID','License Number', 'Car Type'] # Write the header row

create_csv_file()

def save_vehicle_information_to_csv(vehicleID,vehicle_number, vehicle_type):
    # Define the file path for the CSV file
    csv_file = 'vehicle_information.csv'
    # Append the vehicle information to the CSV file
    with open(csv_file, mode='a', newline='') as file:
        writer = csv.writer(file)
        writer.writerow([vehicleID, vehicle_number, vehicle_type])
class_names = {
    0: 'Hatchback',
    1: 'MidSize-SUV',
    3: 'SUV',
    4: 'Sedan'
}
model = YOLO('./best.pt')
# Load the image file
image_path = "./creta.jpg"
frame = cv2.imread(image_path)
# Run YOLOv8 inference on the image
results = model(frame)
cls = []
number_plate = ""
detected_car_type = ""
for result in results:
    boxes = result.boxes.cpu().numpy()
    for i, box in enumerate(boxes):
        if box.conf[0] > 0.4:
            cls = int(box.cls[0])
            if cls == 2:
                r = box.xyxy[0].astype(int)
                crop = frame[r[1]:r[3], r[0]:r[2]]
                cv2.imwrite("numberplate.jpg", crop)
                # Specify the path to the image
                number_plate = 'numberplate.jpg'
            elif cls != 2:
                car_type = class_names.get(cls)
                print(f"Car Type: {car_type}")
                detected_car_type = car_type
# Extract text from the image (implement your extract_text_from_image function)
text = extract_text_from_image(number_plate)
text = str(text)
# Define the pattern for number plate extraction using regular expressions
pattern = r'[A-Z]{2}\s?\d{2}\s?[A-Z]{2}\s?\d{4}'
# Search for the pattern in the extracted text
extracted_number_plates = re.findall(pattern, text)
# Initialize a list to store the results
car_results = []
# Iterate over the extracted number plates
```

```
for plate in extracted_number_plates:
    car_results.append((detected_car_type, plate))
# Print the extracted number plates along with the car type
i = 0   #Vehicle ID
for plate, car_type in car_results:
    i+=1
    save_vehicle_information_to_csv(i,plate,car_type)
import tkinter as tk
import pandas as pd
import tkinter as tk
import csv
def create_grid(frame, rows, columns):
    for i in range(rows):
        frame.rowconfigure(i, weight=1, minsize=50)
        for j in range(columns):
            frame.columnconfigure(j, weight=1, minsize=50)
            cell = tk.Frame(
                master=frame,
                relief=tk.RAISED,
                borderwidth=1
            )
            cell.grid(row=i, column=j, sticky="nsew")
            cell_colors.append(cell)  # Store the cell frame for color assignment
```

```
def assign_vehicle_color(vehicle_type):
    if vehicle_type == "SUV":
        return "red"
    elif vehicle_type == "Sedan":
        return "blue"
    elif vehicle_type == "MidSize-SUV":
        return "green"
    elif vehicle_type == "Hatchback":
        return "yellow"
    else:
        return "white"  # Default color if vehicle type is unknown or not provided
def read_vehicle_data(filename):
    vehicle_data = []
    with open(filename, "r") as file:
        csv_reader = csv.reader(file)
        next(csv_reader)  # Skip the header row
        for row in csv_reader:
            vehicle_id, vehicle_type, vehicle_number = row
            vehicle_data.append((vehicle_id, vehicle_type.strip(), vehicle_number))
    return vehicle_data
def populate_grids(vehicle_data):
    for i, (vehicle_id, vehicle_type, vehicle_number) in enumerate(vehicle_data):
        cell = cell_colors[i % len(cell_colors)]
        color = assign_vehicle_color(vehicle_type)

        # Configure cell's border and background color

        cell.config(bg=color, highlightbackground="black", highlightthickness=1.25)

        # Clear any existing widgets in the cell

        for widget in cell.winfo_children():

            widget.destroy()

        # Create label with custom font style and size

        label_font = ("Sans-Serif", 12, "bold")

        label = tk.Label(cell, text=f"Vehicle ID: {vehicle_id}\n\nType: {vehicle_type}\n\nLicense: {vehicle_number}", bg=color, fg="white", font=label_font)

        label.pack(fill=tk.BOTH, expand=True)

def create_window(vehicle_data):

    window = tk.Tk()

    window.title("Vehicle Data")

    # Calculate the number of grids needed

    num_grids = (len(vehicle_data) + 9) // 10

    for i in range(num_grids):

        # Create the main frame for each grid

        main_frame = tk.Frame(master=window)

        main_frame.pack(fill=tk.BOTH, expand=True)

        # Create the grid

        create_grid(main_frame, 2, 5)

        # Populate the grids with vehicle data

        start_index = i * 10
        end_index = min((i + 1) * 10, len(vehicle_data))
        populate_grids(vehicle_data[start_index:end_index])
    window.mainloop()
# Read vehicle data from CSV file
filename = "vehicle_information.csv"  # Replace with your own filename
vehicle_data = read_vehicle_data(filename)
# List to store cell frames for color assignment
cell_colors = []
# Create the window
create_window(vehicle_data)
cv2.imshow()
print(vehicle_type)
import cv2
import easyocr
import re
import csv
reader = easyocr.Reader(['en'])
video_path = "./entry.mp4"
cap = cv2.VideoCapture(video_path)
vehicle_types = set()
vehicle_numbers = set()
csv_file = open("vehicle_information.csv", "a", newline="")
csv_writer = csv.writer(csv_file)
```

```
while True:
    ret, frame = cap.read()
    if not ret:
        break
    # Run YOLOv8 inference on the frame to get the results
    results = model(frame)
    for result in results:
        boxes = result.boxes.cpu().numpy()
        for i, box in enumerate(boxes):
            if box.conf[0] > 0.4:
                cls = int(box.cls[0])
                if cls == 2:
                    r = box.xyxy[0].astype(int)
                    crop = frame[r[1]:r[3], r[0]:r[2]]
                    cv2.imwrite("numberplate.jpg", crop)
                    # Specify the path to the image
                    number_plate = 'numberplate.jpg'
                elif cls != 2:
                    car_type = class_names.get(cls)
                    print(f"Car Type: {car_type}")
                    detected_car_type = car_type
        # Extract text from the image (implement your extract_text_from_image function)
        text = extract_text_from_image(number_plate)
        text = str(text)

        # Define the pattern for number plate extraction using regular expressions

        pattern = r'[A-Z]{2}\s?\d{2}\s?[A-Z]{2}\s?\d{4}'

        # Search for the pattern in the extracted text

        extracted_number_plates = re.findall(pattern, text)

        # Iterate over the extracted number plates

        for plate in extracted_number_plates:

            # Add the vehicle type and number to the respective sets

            vehicle_types.add(detected_car_type)

            vehicle_numbers.add(plate)

cap.release()

csv_file.close()

with open("vehicle_information.csv", "w", newline="") as csv_file:

    csv_writer = csv.writer(csv_file)

    for vehicle_type, vehicle_number in zip(vehicle_types, vehicle_numbers):

        csv_writer.writerow([vehicle_type, vehicle_number])

for plate, car_type in zip(vehicle_types, vehicle_numbers):

    print(f"This car with number: {plate} is a {car_type} type.")

import tkinter as tk

import csv

class ParkingLot:

    def __init__(self, capacity):

        self.capacity = capacity
        self.spaces = [None] * capacity

class Car:

    def __init__(self, car_type, car_number):

        self.car_type = car_type

        self.car_number = car_number

def allocate_parking_space(car_type, car_number, parking_lots):

    for parking_lot in parking_lots:

        if parking_lot.spaces.count(None) == 0:

            continue  # Move to the next parking lot if the current one is full

        for i, space in enumerate(parking_lot.spaces):

            if space is None:

                if car_type == 'SUV':

                    suv_count = sum(1 for s in parking_lot.spaces if s and s.car_type == 'SUV')

                    if suv_count >= 0.4 * parking_lot.capacity:

                        continue  # SUV limit reached in this parking lot

                if car_type == 'SUV' and any(s and s.car_type == 'SUV' for s in
parking_lot.spaces[i - 1:i + 2]):

                    continue  # Don't allow two SUVs to be near

                if car_type in ['Sedan', 'Hatchback']:

                    neighboring_count = 0

                    for j in range(i - 1, i + 2):

                        if 0 <= j < parking_lot.capacity and parking_lot.spaces[j] and
parking_lot.spaces[

                            j].car_type in ['Sedan', 'Hatchback']:
```

```
                    neighboring_count += 1

                if neighboring_count >= 2:

                    continue  # Limit reached for Sedans and Hatchbacks

            parking_lot.spaces[i] = Car(car_type, car_number)

            return f'Car {car_number} parked at Lot {parking_lots.index(parking_lot) + 1}, Space {i + 1}'

    return 'No available parking space'

def read_vehicle_data(filename):

    vehicle_data = []

    with open(filename, "r") as file:

        csv_reader = csv.reader(file)

        next(csv_reader)  # Skip the header row

        for row in csv_reader:

            vehicle_id, vehicle_number, vehicle_type = row

            vehicle_data.append((vehicle_id, vehicle_number, vehicle_type.strip()))

    return vehicle_data

def create_grid(frame, rows, columns):

    for i in range(rows):

        frame.rowconfigure(i, weight=1, minsize=50)

        for j in range(columns):

            frame.columnconfigure(j, weight=1, minsize=50)

            cell = tk.Frame(

                master=frame,

                relief=tk.RAISED,
            borderwidth=1

          )

          cell.grid(row=i, column=j, sticky="nsew")

          cell_frames.append(cell)

def populate_grids(vehicle_data):

    for i, (vehicle_id, _, vehicle_type) in enumerate(vehicle_data):

        cell = cell_frames[i % len(cell_frames)]

        color = assign_vehicle_color(vehicle_type)

        cell.config(bg=color)

        # Clear any existing widgets in the cell

        for widget in cell.winfo_children():

            widget.destroy()

        label_font = ("Arial", 12, "bold")

        label = tk.Label(cell, text=f"Vehicle ID: {vehicle_id}\nType: {vehicle_type}",
bg=color, fg="white", font=label_font)

        label.pack(fill=tk.BOTH, expand=True)

def assign_vehicle_color(vehicle_type):

    colors = {'SUV': 'red', 'Mid Size SUV': 'purple', 'Sedan': 'green', 'Hatchback': 'blue'}

    return colors.get(vehicle_type, 'white')

def load_vehicle_data():

    filename = "input.csv"  # Update with your CSV file path

    vehicle_data = read_vehicle_data(filename)

    populate_grids(vehicle_data)

# Initialize Tkinter window
```

```
window = tk.Tk()

window.title("Parking Lot")

window.geometry("800x600")

# Create main frame

main_frame = tk.Frame(master=window)

main_frame.pack(fill=tk.BOTH, expand=True)

# Create cell frames

cell_frames = []

create_grid(main_frame, 2, 5)

# Load vehicle data and populate grids

load_vehicle_data()

# Add a button to reload vehicle data

button_frame = tk.Frame(master=window)

button_frame.pack(pady=10)

reload_button = tk.Button(button_frame, text="Reload Vehicle Data", command=load_vehicle_data)

reload_button.pack()

# Start the Tkinter event loop

window.mainloop()

def populate_grids(vehicle_data):

    for i, (vehicle_id, _, vehicle_type) in enumerate(vehicle_data):

        allocated = False

        for cell in cell_frames:

            color = assign_vehicle_color(vehicle_type)
            if not cell.winfo_children():  # Check if cell is empty

                row, col = cell.grid_info()["row"], cell.grid_info()["column"]

                if can_allocate_vehicle(row, col, vehicle_type):

                    cell.config(bg=color)

                    label_font = ("Arial", 12, "bold")

                    label = tk.Label(cell, text=f"Vehicle ID: {vehicle_id}\nType: {vehicle_type}",
bg=color, fg="white", font=label_font)

                    label.pack(fill=tk.BOTH, expand=True)

                    allocated = True

                    break

        if not allocated:

            # Create a new grid layout and allocate the vehicle

            create_grid(main_frame, 2, 5)

            cell = cell_frames[-1]

            color = assign_vehicle_color(vehicle_type)

            cell.config(bg=color)

            label_font = ("Arial", 12, "bold")

            label = tk.Label(cell, text=f"Vehicle ID: {vehicle_id}\nType: {vehicle_type}",
bg=color, fg="white", font=label_font)

            label.pack(fill=tk.BOTH, expand=True)

def can_allocate_vehicle(row, col, vehicle_type):

    if vehicle_type == 'SUV':

        suv_count = sum(1 for cell in cell_frames if cell.winfo_children() and
cell.winfo_children()[0]['text'].split("\n")[1] == 'SUV')

        if suv_count >= 0.4 * len(cell_frames):
            return False

        if row > 0 and cell_frames[(row - 1) * 5 + col].winfo_children() and cell_frames[(row - 1) * 5 + col].winfo_children()[0]['text'].split("\n")[1] == 'SUV':

            return False

        if row < 1 and cell_frames[(row + 1) * 5 + col].winfo_children() and cell_frames[(row + 1) * 5 + col].winfo_children()[0]['text'].split("\n")[1] == 'SUV':

            return False

        if col > 0 and cell_frames[row * 5 + col - 1].winfo_children() and cell_frames[row * 5 + col - 1].winfo_children()[0]['text'].split("\n")[1] == 'SUV':

            return False

        if col < 4 and cell_frames[row * 5 + col + 1].winfo_children() and cell_frames[row * 5 + col + 1].winfo_children()[0]['text'].split("\n")[1] == 'SUV':

            return False

    else:

        neighboring_count = 0

        if row > 0 and cell_frames[(row - 1) * 5 + col].winfo_children() and cell_frames[(row - 1) * 5 + col].winfo_children()[0]['text'].split("\n")[1] in ['Sedan', 'Hatchback']:

            neighboring_count += 1

        if row < 1 and cell_frames[(row + 1) * 5 + col].winfo_children() and cell_frames[(row + 1) * 5 + col].winfo_children()[0]['text'].split("\n")[1] in ['Sedan', 'Hatchback']:

            neighboring_count += 1

        if col > 0 and cell_frames[row * 5 + col - 1].winfo_children() and cell_frames[row * 5 + col - 1].winfo_children()[0]['text'].split("\n")[1] in ['Sedan', 'Hatchback']:

            neighboring_count += 1

        if col < 4 and cell_frames[row * 5 + col + 1].winfo_children() and cell_frames[row * 5 + col + 1].winfo_children()[0]['text'].split("\n")[1] in ['Sedan', 'Hatchback']:

            neighboring_count += 1
        if neighboring_count >= 2:
            return False
    return True
def load_vehicle_data():
    filename = "input.csv"  # Update with your CSV file path
    vehicle_data = read_vehicle_data(filename)
    populate_grids(vehicle_data)
# Initialize Tkinter window
window = tk.Tk()
window.title("Parking Lot")
window.geometry("800x600")
# Create main frame
main_frame = tk.Frame(master=window)
main_frame.pack(fill=tk.BOTH, expand=True)
# Create cell frames
cell_frames = []
create_grid(main_frame, 2, 5)
# Load vehicle data and populate grids
load_vehicle_data()
# Add a button to reload vehicle data
button_frame = tk.Frame(master=window)
button_frame.pack(pady=10)
reload_button = tk.Button(button_frame, text="Reload Vehicle Data", command=load_vehicle_data)
```

```
reload_button.pack()

# Start the Tkinter event loop

window.mainloop()


import pandas as pd
import tkinter as tk
from tkinter import filedialog
class ParkingLot:
   def __init__(self, capacity):
      self.capacity = capacity
      self.spaces = [None] * capacity
class Car:
   def __init__(self, car_type, car_number, car_status):
      self.car_type = car_type
      self.car_number = car_number
      self.car_status = car_status
def process_csv(file_path):
   df = pd.read_csv(file_path)
   car_inputs = df[['Car Type', 'Number', 'Car Status']].values.tolist()
   return car_inputs
def allocate_parking_space(car_type, car_number, car_status, parking_lots):
   for parking_lot in parking_lots:
      if parking_lot.spaces.count(None) == 0:
         continue  # Move to the next parking lot if the current one is full
      for i, space in enumerate(parking_lot.spaces):
         if space is None:
            if car_type == 'SUV':
               suv_count = sum(1 for s in parking_lot.spaces if s and s.car_type == 'SUV')
               if suv_count >= 0.4 * parking_lot.capacity:
                  continue  # SUV limit reached in this parking lot
            if car_type == 'SUV' and any(s and s.car_type == 'SUV' for s in
parking_lot.spaces[i - 1:i + 2]):
               continue  # Don't allow two SUVs to be near
            if car_type in ['Sedan', 'Hatchback']:
               neighboring_count = 0
               for j in range(i - 1, i + 2):
                  if 0 <= j < parking_lot.capacity and parking_lot.spaces[j] and
parking_lot.spaces[
                     j].car_type in ['Sedan', 'Hatchback']:
                     neighboring_count += 1
               if neighboring_count >= 2:
                  continue  # Limit reached for Sedans and Hatchbacks
            if car_status == 1:
               parking_lot.spaces[i] = Car(car_type, car_number, car_status)
            else:
               parking_lot.spaces[i] = None
            return f'Car {car_number} parked at Lot {parking_lots.index(parking_lot) + 1},
Space {i + 1}'
   return 'No available parking space'
def create_parking_gui(parking_lots):
   root = tk.Tk()
   root.title("Parking Lot Status")
   # Color map for car types
   color_map = {'SUV': 'red', 'MidSize-SUV': 'purple', 'Sedan': 'green', 'Hatchback': 'blue'}
   # Create a tkinter frame for the labels
   labels_frame = tk.Frame(root, padx=10, pady=10)
   labels_frame.pack(expand=True, fill=tk.BOTH)
   # Update the labels with parking lot status
   def update_labels():
      for row, parking_lot in enumerate(parking_lots):
         for col, space in enumerate(parking_lot.spaces[:5]):
            if space is None:
               cell_value = ""
               cell_color = "white"
            else:
               cell_value = space.car_number
               cell_color = color_map[space.car_type]

            label = labels[row][col]
            label.config(text=cell_value, bg=cell_color)
            label.update()
   # Create the labels dynamically
   # Create the labels dynamically
   labels = []
   lot_number = 1  # Initialize the lot number
   for row in range(5):
      row_labels = []
      for col in range(5):
         label = tk.Label(labels_frame, text=str(lot_number), width=25, height=5,
relief=tk.RAISED, bd=1,
                  font=("Times New Roman", 12, 'bold'), fg='white')
         label.grid(row=row, column=col, padx=5, pady=5, ipadx=2, ipady=2, sticky="nsew")
         row_labels.append(label)
         #row_labels.append(str(lot_number))
         labels_frame.grid_columnconfigure(col, weight=1)
         #lot_number += 1  # Increment the lot number
      labels.append(row_labels)
```

```
        labels_frame.grid_rowconfigure(row, weight=1)
    # Add numbering to the labels
    for i, row_labels in enumerate(labels):
        for j, label in enumerate(row_labels):
            lot_number_label = tk.Label(label, text=str(lot_number), font=("Times New
Roman", 12, 'bold'), fg='white')
            lot_number_label.pack()
            lot_number_label.config(fg='black')  # Change the color to black
            lot_number += 1
    update_labels()
    # Create a tkinter frame for the legend
    legend_frame = tk.Frame(root)
    legend_frame.pack(side=tk.RIGHT, fill=tk.X)
    # Create the legend labels
    for car_type, color in color_map.items():
        legend_label = tk.Label(legend_frame, text=car_type, bg=color, fg='white', width=15,
padx=5, pady=5,
                                font=("Helvetica", 12))
        legend_label.pack(side=tk.LEFT)
    # Create a tkinter frame for the buttons
    button_frame = tk.Frame(root)
    button_frame.pack(side=tk.BOTTOM, pady=10, fill=tk.X)
    # Add a button to reload the CSV file
    def reload_csv():
        file_path = filedialog.askopenfilename(filetypes=[("CSV files", "*.csv")])
        if file_path:
            car_inputs = process_csv(file_path)
            for car_type, car_number, car_status in car_inputs:
                result = allocate_parking_space(car_type, car_number, car_status, parking_lots)
                print(result)
            update_labels()
    reload_button = tk.Button(button_frame, text="Reload CSV", command=reload_csv)
    reload_button.pack(side=tk.LEFT, padx=10)
    # Add a button to close the GUI
    close_button = tk.Button(button_frame, text="Close", command=root.quit)
    close_button.pack(side=tk.LEFT, padx=10)
    root.mainloop()
# Prompt the user to select a CSV file
file_path = filedialog.askopenfilename(filetypes=[("CSV files", "*.csv")])
if file_path:
    car_inputs = process_csv(file_path)
    parking_lots = [ParkingLot(25) for _ in range(3)]
    for car_type, car_number, car_status in car_inputs:
        for parking_lot in parking_lots:
            for space in parking_lot.spaces:
```

```
            if space and space.car_number == car_number:
                if car_status == 0:
                    parking_lot.spaces.remove(space)
                break
    else:
        result = allocate_parking_space(car_type, car_number, car_status, parking_lots)
        print(result)
create_parking_gui(parking_lots)
```