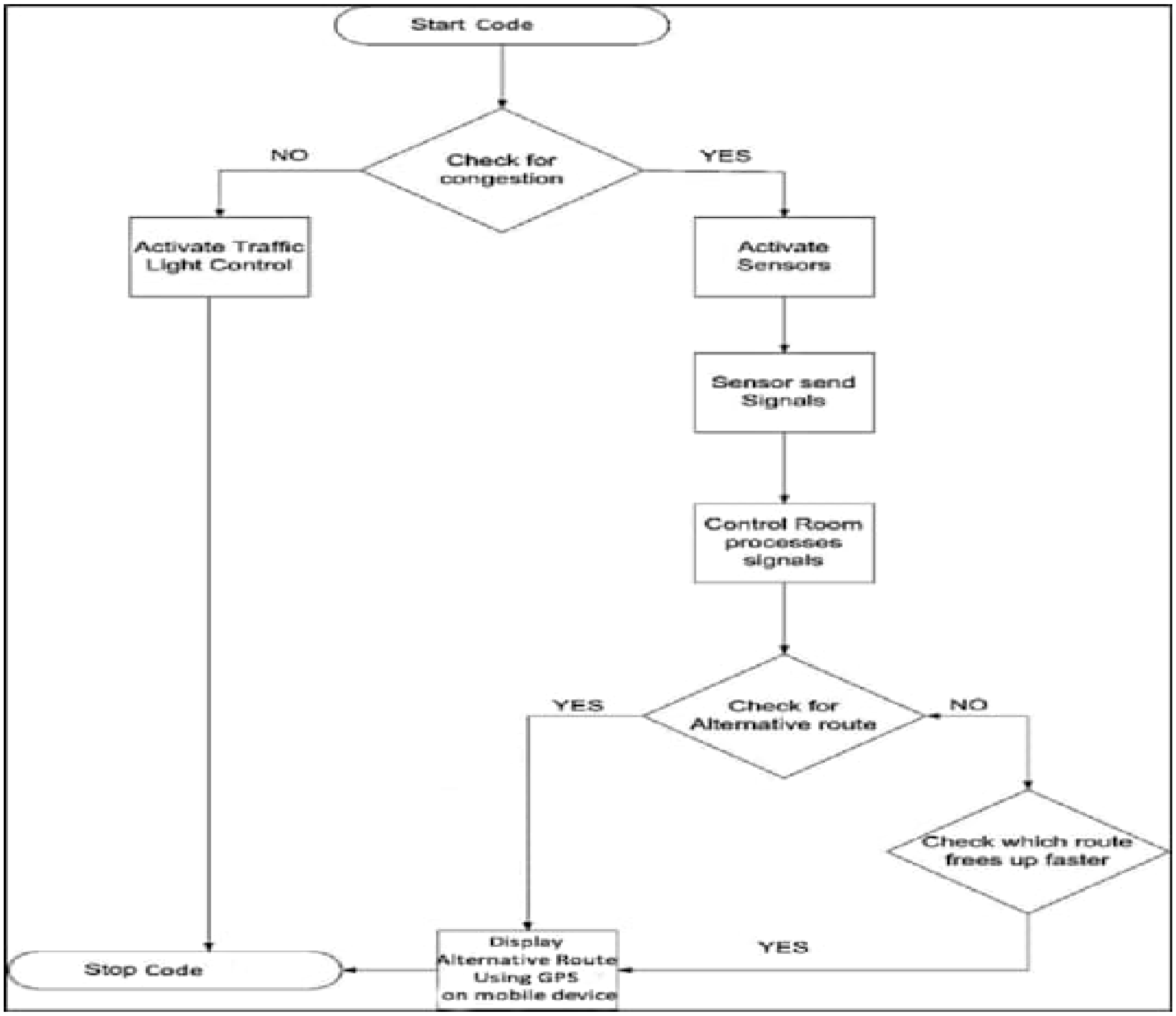
Start Code
Check for congestion
NO
YES
Activate Traffic Light Control
Activate Sensors
Sensor send Signals
Control Room processes signals
Check for Alternative route
YES
NO
Check which route frees up faster
YES
Display Alternative Route Using GPS on mobile device
Stop Code

A Traffic Management System (TMS) typically consists of various components and subsystems that work together to manage and control traffic flow. Here's a simplified block diagram explanation of a TMS:

1. *Traffic Sensors*: These are various sensors placed on the road, such as cameras, inductive loops, or radar detectors. They collect data on traffic conditions, including vehicle presence, speed, and volume.

2. *Data Collection and Processing*: The data from the sensors are transmitted to a central control system. This system collects, processes, and analyzes the data to understand the current traffic situation.

3. *Traffic Control Center*: This is where traffic controllers and operators monitor the data from the sensors in real-time. They use this information to make decisions and control traffic signals, variable message signs, and other devices.

4. *Traffic Signals and Signs*: Traffic signals at intersections can be controlled in real-time based on traffic conditions. Variable message signs along roads can display information and warnings to drivers.

5. *Communication Network*: A reliable communication network, often using fiber optics or wireless technology, connects the various components of the system, ensuring data is transmitted quickly and without interruption.

6. *Traffic Management Algorithms*: These are the software algorithms that process the data and make decisions about traffic signal timing, lane control, and other actions to optimize traffic flow and safety.

7. *Emergency Response Integration*: TMS can be integrated with emergency services to ensure rapid response to incidents, such as accidents or road closures.

8. *Driver Information Systems*: TMS can provide real-time information to drivers through apps, websites, or electronic signs, helping them make informed decisions about their routes.

9. *Traffic Data Storage and Analysis*: Historical traffic data can be stored and analyzed for long-term planning and optimization of the road network.

10. *Maintenance and Diagnostics*: Systems for monitoring and maintaining the sensors, control equipment, and other components to ensure they are functioning correctly.

11. *Feedback Loop*: Continuous monitoring and feedback from the system allow for adjustments and improvements to traffic management strategies.

This block diagram shows the major components of a Traffic Management System, which collectively work to enhance traffic safety, reduce congestion, and improve the overall efficiency of road networks. The specifics of each TMS can vary depending on the scale and complexity of the transportation system it serves.

```
#import lib files


from tracking.centroidtracker import CentroidTracker

from tracking.trackableobject import TrackableObject

import tensornets as nets

import cv2

import numpy as np

import time

import dlib

import tensorflow.compat.v1 as tf

import os

import threading


def countVehicles(param):

# param -> path of the video

            # list -> number of vehicles will be written in the list

            # index ->Index at which data has to be written

tf.disable_v2_behavior()

            # Image size must be '416x416' as YoloV3 network expects that specific image size as input

img_size = 416

   inputs = tf.placeholder(tf.float32, [None, img_size, img_size, 3])

   model = nets.YOLOv3COCO(inputs, nets.Darknet19)


ct = CentroidTracker(maxDisappeared=5, maxDistance=50) # Look into 'CentroidTracker' for further info
about parameters

   trackers = [] # List of all dlib trackers

   trackableObjects = {} # Dictionary of trackable objects containing object's ID and its' corresponding
centroid/s

   skip_frames = 10 # Numbers of frames to skip from detecting
```

```
    confidence_level = 0.40 # The confidence level of a detection

    total = 0 # Total number of detected objects from classes of interest

    use_original_video_size_as_output_size = True # Shows original video as output and not the 416x416
image that is used as yolov3 input (NOTE: Detection still happens with 416x416 img size but the output
is displayed in original video size if this parameter is True)


    video_path = os.getcwd() + param # "/videos/4.mp4"

    video_name = os.path.basename(video_path)


# print("Loading video {video_path}...".format(video_path=video_path))

if not os.path.exists(video_path):

        print("File does not exist. Exited.")

        exit()


# YoloV3 detects 80 classes represented below

all_classes = ["person", "bicycle", "car", "motorbike", "aeroplane", "bus", "train", "truck", \

                "boat", "traffic light", "fire hydrant", "stop sign", "parking meter", "bench", \

                "bird", "cat", "dog", "horse", "sheep", "cow", "elephant", "bear", "zebra", "giraffe", \

                "backpack", "umbrella", "handbag", "tie", "suitcase", "frisbee", "skis", "snowboard", \

                "sports ball", "kite", "baseball bat", "baseball glove", "skateboard", "surfboard", \

                "tennis racket", "bottle", "wine glass", "cup", "fork", "knife", "spoon", "bowl", "banana", \

                "apple", "sandwich", "orange", "broccoli", "carrot", "hot dog", "pizza", "donut", "cake", \

                "chair", "sofa", "pottedplant", "bed", "diningtable", "toilet", "tvmonitor", "laptop", "mouse", \

                "remote", "keyboard", "cell phone", "microwave", "oven", "toaster", "sink", "refrigerator", \

                "book", "clock", "vase", "scissors", "teddy bear", "hair drier", "toothbrush"]


# Classes of interest (with their corresponding indexes for easier looping)

classes = { 1 : 'bicycle', 2 : 'car', 3 : 'motorbike', 5 : 'bus', 7 : 'truck' }
```

```
    with tf.Session() as sess:
        sess.run(model.pretrained())
        cap = cv2.VideoCapture(video_path)

# Get video size (just for log purposes)
width =  int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
        height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))

# Scale used for output window size and net size
width_scale = 1
        height_scale = 1

        if use_original_video_size_as_output_size:
            width_scale = width / img_size
            height_scale = height / img_size

        def drawRectangleCV2(img, pt1, pt2, color, thickness, width_scale=width_scale,
height_scale=height_scale):
            point1 = (int(pt1[0] * width_scale), int(pt1[1] * height_scale))
            point2 = (int(pt2[0] * width_scale), int(pt2[1] * height_scale))
            return cv2.rectangle(img, point1, point2, color, thickness)

        def drawTextCV2(img, text, pt, font, font_scale, color, lineType, width_scale=width_scale,
height_scale=height_scale):
            pt = (int(pt[0] * width_scale), int(pt[1] * height_scale))
            cv2.putText(img, text, pt, font, font_scale, color, lineType)

        def drawCircleCV2(img, center, radius, color, thickness, width_scale=width_scale,
height_scale=height_scale):
            center = (int(center[0] * width_scale), int(center[1] * height_scale))
```

```
          cv2.circle(img, center, radius, color, thickness)
# Python 3.5.6 does not support f-strings (next line will generate syntax error)
                    #print(f"Loaded {video_path}. Width: {width}, Height: {height}")
                    # print("Loaded {video_path}. Width: {width}, Height:
{height}".format(video_path=video_path, width=width, height=height))
skipped_frames_counter = 0


        while(cap.isOpened()):
          try :
            ret, frame = cap.read()
            img = cv2.resize(frame, (img_size, img_size))
          except:
            print(total_str)



          output_img = frame if use_original_video_size_as_output_size else img


          tracker_rects = []


          if skipped_frames_counter == skip_frames:


# Detecting happens after number of frames have passes specified by 'skip_frames' variable value
                                        # print("[DETECTING]")
trackers = []
            skipped_frames_counter = 0 # reset counter


            np_img = np.array(img).reshape(-1, img_size, img_size, 3)


            start_time=time.time()
            predictions = sess.run(model.preds, {inputs: model.preprocess(np_img)})
# print("Detection took %s seconds" % (time.time() - start_time))


                                            # model.get_boxes returns a 80 element array containing information
about detected classes
                                            # each element contains a list of detected boxes, confidence level ...
detections = model.get_boxes(predictions, np_img.shape[1:3])
            np_detections = np.array(detections)
# Loop only through classes we are interested in
for class_index in classes.keys():
               local_count = 0
               class_name = classes[class_index]


# Loop through detected infos of a class we are interested in
for i in range(len(np_detections[class_index])):
                  box = np_detections[class_index][i]


                  if np_detections[class_index][i][4] >= confidence_level:
# print("Detected ", class_name, " with confidence of ", np_detections[class_index][i][4])
local_count += 1
                     startX, startY, endX, endY = box[0], box[1], box[2], box[3]


                     drawRectangleCV2(output_img, (startX, startY), (endX, endY), (0, 255, 0), 1)
                     drawTextCV2(output_img, class_name, (startX, startY), cv2.FONT_HERSHEY_SIMPLEX, .5,
(0, 0, 255), 1)


                                                                             # Construct a dlib rectangle object from the
bounding box coordinates and then start the dlib correlation
tracker = dlib.correlation_tracker()
                     rect = dlib.rectangle(int(startX), int(startY), int(endX), int(endY))
```

```
                        tracker.start_track(img, rect)


                                                                    # Add the tracker to our list of trackers so we
can utilize it during skip frames
# Write the total number of detected objects for a given class on this frame
                                                    # print(class_name," : ", local_count)
else:
# If detection is not happening then track previously detected objects (if any)
                                            # print("[TRACKING]")
skipped_frames_counter += 1
# Increase the number frames for which we did not use detection
                                            # Loop through tracker, update each of them and display their rectangle
for tracker in trackers:
            tracker.update(img)
            pos = tracker.get_position()



                                                    # Unpack the position object
startX = int(pos.left())
            startY = int(pos.top())
            endX = int(pos.right())
            endY = int(pos.bottom())


                                                    # Add the bounding box coordinates to the tracking rectangles
list
            tracker_rects.append((startX, startY, endX, endY))
                                                    # Draw tracking rectangles
            drawRectangleCV2(output_img, (startX, startY), (endX, endY), (255, 0, 0), 1)
                                    # Use the centroid tracker to associate the (1) old object centroids with (2) the
newly computed object centroids
```

```
        objects = ct.update(tracker_rects)

                          # Loop over the tracked objects

        for (objectID, centroid) in objects.items():

                                      # Check to see if a trackable object exists for the current object ID

to = trackableObjects.get(objectID, None)


          if to is None:

                                               # If there is no existing trackable object, create one

to = TrackableObject(objectID, centroid)

          else:

             to.centroids.append(centroid)


                                               # If the object has not been counted, count it and mark it as
counted

if not to.counted:

               total += 1

               to.counted = True

                                      # Store the trackable object in our dictionary

          trackableObjects[objectID] = to

                                      # Draw both the ID of the object and the centroid of the object on the
output frame

object_id = "ID {}".format(objectID)

          drawTextCV2(output_img, object_id, (centroid[0] - 10, centroid[1] - 10),
cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 1)

          drawCircleCV2(output_img, (centroid[0], centroid[1]), 2, (0, 255, 0), -1)


                                      # Display the total count so far

total_str = str(total)

          drawTextCV2(output_img, total_str, (10, 30), cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0, 0, 255), 2)
```

```
                                    # Display the current frame (with all annotations drawn up to this point)
cv2.imshow(video_name, output_img)

      key = cv2.waitKey(1) & 0xFF
      if key  == ord('q'): # QUIT (exits)
        break
      elif key == ord('p'):
        cv2.waitKey(0) # PAUSE (Enter any key to continue)

  cap.release()
  cv2.destroyAllWindows()
  print("Exited")

  """
  function which will run our code

  will write the number of veicles in the list provided
  """

if __name__ == "__main__":



  countVehicles("/videos/test.mp4")

       # Logic for setting the time for each signal
```