EE417 COMPUTER VISION TERM PROJECT

GOKTUG KORKULU
27026

FREE PARKING SPACE COUNTER

GOKTUG KORKULU

INTRODUCTION

In this project, I will create a free parking space detector and counter. First, I have one video and one image which is extracted from one of the frame of that video. My general purpose is to be able to detect and count the number of free spaces on any parking lot whose video is taken in birdeye view. Basically what I will do is that, I will try to approach this problem with methods as robust as possible. However, in my opinion since some of the requirements demand more than what I can give within the scope of this EE417 - COMPUTER VISION course, I will do some manual work as well. But it is better to keep in mind that, the manual work that I will be doing is more like other Computer Science related courses' topic. The counter/ detector that I will try to develop will be as less sensitive as possible to the minor changes inside the parking lots such as people walking inside or trolleys. So, let's talk about the methods that I will be using in order to achieve my goal.

METHODS

We are going to start with writing a program that will allow us to select/deselect the parking spaces first. We will put the selected parking spaces in a list, then will grab and take it to our main code that will count and show the free parking spaces. In short words, one python file for selecting, on for running. Let's start with the one that we select and store the parking spaces

FreeSpaceFinder.py

STEP 1

First, lets import the packages that we will be utilizing to achieve what we want in this python file.

```
import cv2
import pickle
```

Cv2 is computer vision package for python that we will be using a lot. In addition to that, pickle is the package that we will be using to save all the locations of the parking spaces and then bring it to out main code for further development.

Now we will import our image and then show it on a screen so that we can work on it. However, what is different than we do regularly is that, we will put these commands in an infinite while loop so that every iteration of while loop the altered image will be read and shown until we terminate the program. Below code corresponds what is explained here.

```
while True:
    img = cv2.imread('carParkImg.png')
    cv2.imshow('image', img)
    cv2.waitKey(1)
```

cv2.waitkey(1) makes the main thread to wait for 1 millisecond before looping another iteration. The resulting will be as follows.

The above image is read and shown every 1ms to a window

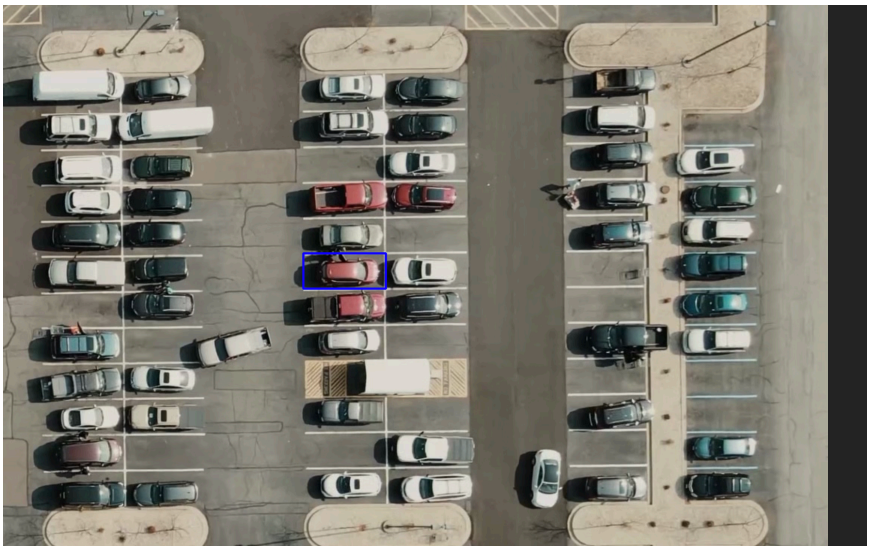
STEP 2

What we need now is, we ought to get the average width and the average height values of parking spaces in terms of pixels, from the given image.

First, let's create a rectangle and try to plate that to one of the parking lots as precise and fitted as possible. In order to achieve this goal, I will need to utilize cv2.rectangle() built-in computer vision method which creates rectangles by taking left-upper and right-lower corners as inputs.

```
cv2.rectangle(img, (400,330),(510,377),
(255,0,0),2)
# width:110, height:47
```

Note that the first parameter *img* is the input image which the rectangle will be drawn on top of it, second parameter *(400,330)* is any pixel coordinates that corresponds to left-upper corner of the rectangle to be drawn and similarly third parameter *(510,377)* is any pixel coordinates that corresponds to right-lower corner of the rectangle to be drawn. The fourth and fifth parameters *(255,0,0)* and *2* are the color and the thickness of rectangle borders respectively.



The resulting output of the given code above is shown here. Notice the blue colored rectangle that is fitted to an arbitrary parking lot.

I will elaborate more here. What we actually need is just width and height values in pixels. That's why I just put

random initial coordinate values and tried to fit the rectangle by adjusting the parameters I have mentioner earlier to an arbitrary parking lot that can be seen in the previous image.

Based on the single rectangle that I fitted to a parking lot as much precise as possible, the width and the height values are 110 and 47 pixels respectively. I will assign these values to parameters called *width* and *height* and use it to create remaining rectangles by using these values. In other words, once we had the width and the height values, then all we need will be the initial position's coordinates (i.e. left-upper corners) of each parking spaces.

STEP 3

As mentioned recently, I will need to keep the initial positions of the rectangles so that I can create rectangles based on those values.

Now, we will create an array which keeps the left-upper corner coordinates of EACH rectangle that supposed to fit precisely to parking spaces in the image. Let's call this array *startingCoordinates*.

```
startingCoordinates = []
```

In order to select and put each starting coordinates of parking spaces, we need to select them manually by ourselves. Therefore, we will need to use mouse click listener. For this purpose, we will utilize *setMouseCallback()* built-in function from computer vision package.

```
cv2.setMouseCallback("image",
coordinateSelector)
```

The first parameter *'image'* is the image that we will listen for the mouse clicks. This name should be exactly same with the one that I put when using vs2.imshow() function. The second parameter *'coordinateSelector'* is the function that will be called on click event.

Let's dive into coordinateSelector() function and explain how does it help us to keep the initial coordinate values of each rectangle.

```
def coordinateSelector(events, x, y, flags, parameters):
    if events == cv2.EVENT_LBUTTONDOWN:
        startingCoordinates.append((x, y))
    if events == cv2.EVENT_RBUTTONDOWN:
        i = -1;
        for coordinate in startingCoordinates:
            i = i + 1
            xi, yi = coordinate
            if xi < x < xi + width and yi < y < yi + height:
                startingCoordinates.pop(i)
    with open('finalCoordinates', 'wb') as f:
        pickle.dump(startingCoordinates, f)
```

Inside the coordinateSelector() function, there are 5 parameters which the first 3 are important for us. First parameter *events* is the information that is sent by the setMouseCallback() function which we will use it to check the mouse click event type, i.e. if it is right click or left click. Second and Third parameters *x* and *y* are x and y coordinates of the point that is clicked on the image.

Initially, we check if the clicked button on the mouse is the left button. If so, we will append that exact pixel coordinate to my startingCoordinates[] array.

Similarly, in case of wrongly selected rectangle or clicks happened by mistake, we need to remove and delete the wrong coordinates from the array by making right click on the mouse. To do that, we are going to check if the clicked coordinates are in between any rectangle that is already

created ant put inside the array. I do this check in the if statement that is inside the for loop. In other words, what we do here is checking whether the clicked coordinate pixels is inside the rectangle borders and if it is, then we remove/pop that rectangle by accessing it from the index value i.

STEP 4

Let's focus on one important issue now. As you might have noticed, we only create these coordinates and store them in a local array called startingCoordinates[]. However, whenever we terminate the FreeSpaceFinder.py program, all data we had just created and stored is wiped out. In order to keep the seating point coordinates in a global item/folder and utilize them after the termination of the FreeSpaceFinder.py program, let's use pickle object.

First, every time that I insert or remove a coordinate point to my list, I will adjust the resulting outcome to my pickle object. How we do this is below.

```
with open('finalCoordinates', 'wb') as f:
        pickle.dump(startingCoordinates, f)
```

The first parameter of open() function is for declaring the name of the global file that will be storing the coordinates data. Second parameter is for granting writing permission to that file.

The first parameter of dump() function is the name of the local array that I was keeping the coordinates data and the second one is the file that we will write the array data to on top of it.

Now, the problem is that every time that I run the program, f file will be overwritten rather than retrieving the previous data from the file and merging it with the new changes on the list. In order to avoid this problem, we should put a bunch of code that checks if *finalCoordinates* file is already exists and according to that result, will make changes. Below is the code of this idea.

```
try:
    with open('finalCoordinates', 'rb') as f:
        startingCoordinates = pickle.load(f)
except:
    startingCoordinates = []
```

What this code does is, it checks if the file called *finalCoordinates* is already exists by checking if it can be opened. If yes, we initialize our *startingCoordinates* by loading the data from that file and if no, we initialize our array as an empty list.

Now, when you make changes on the startingCoordiantes list then terminate the program; after re running this program, all the changes will be saved and retrieved.

Finally, what we achieved so far in FreeSpaceFinder.py file is that; we selected and fitted all the parking lots into rectangles and saved these informations to *finalCoordinates* file. Now, we can utilize this file to start working on the most important part, the main.py file.

Just before stepping further and explain the main.py file, I wanted to visualize what we have achieved in order for better understanding. Thanks to below code, we can see all

the rectangles whose starting points are stored in the *finalCoordinates* and observe the live changes on the visualisation.

```
for coordinate in startingCoordinates:
    cv2.rectangle(img, coordinate,
    (coordinate[0] + width, coordinate[1] +
    height), (255, 0, 0), 2)
```

The demonstration video link is attached above, all I have done has been visualized here:

https://drive.google.com/file/d/175EM-WANe4maxVwICvSKYyhWcDA9uWUD/view?usp=sharing

Main.py

Let's import the libraries that we will be using in this program. I imported numpy library to utilize it when I will need to create a kernel matrix later on. I imported cvzone to use it later in step5 to put text into the rectangles.

```
import cv2
import pickle
import cvzone
import numpy as np
```

STEP 1

The first thing we need to do after importing the libraries is to open up our video that is called *carPark.mp4*.

```
videoCapture =
cv2.VideoCapture('carPark.mp4')
while True:
    success, image = videoCapture.read()
    cv2.imshow("Image", image)
    cv2.waitKey(1)
```

Now we are able to run the video but when the code terminates, video disappears from the screen. What we need is to loop the video so that we can make observations clearly and in a continuous manner. To do that, we write the below code inside the while loop.

The verbal explanation of the code below is; If the current running frame of the video is equal to the total frame of the video, then we set the current frame of the video to zero so that we run the video nonstop.

```
if videoCapture.get(cv2.CAP_PROP_POS_FRAMES) ==
videoCapture.get(cv2.CAP_PROP_FRAME_COUNT):
   videoCapture.set(cv2.CAP_PROP_POS_FRAMES, 0)
```

STEP 2

Let's import all the positions we have (i.e. starting coordinates of rectangles) in the file called *finalCoordinates*. We can use the same method of loading as we did in the previous part. Unlikely, this time we do not need to check if the file exists or not since we know it is for sure. That's why, let's add below code to our program.

```
with open('finalCoordinates', 'rb') as f:
    startingCoordinates = pickle.load(f)
```

Now, within out startingCoordinates[] list, we can display rectangles by the same code we have used in the previous part as well.

```
for coordinate in startingCoordinates:
    cv2.rectangle(img, coordinate,
    (coordinate[0] + width, coordinate[1] +
    height), (255, 0, 0), 2)
```

Now, to wrap up what we have done so far; we are displaying the rectangles on the infinitely looped video.

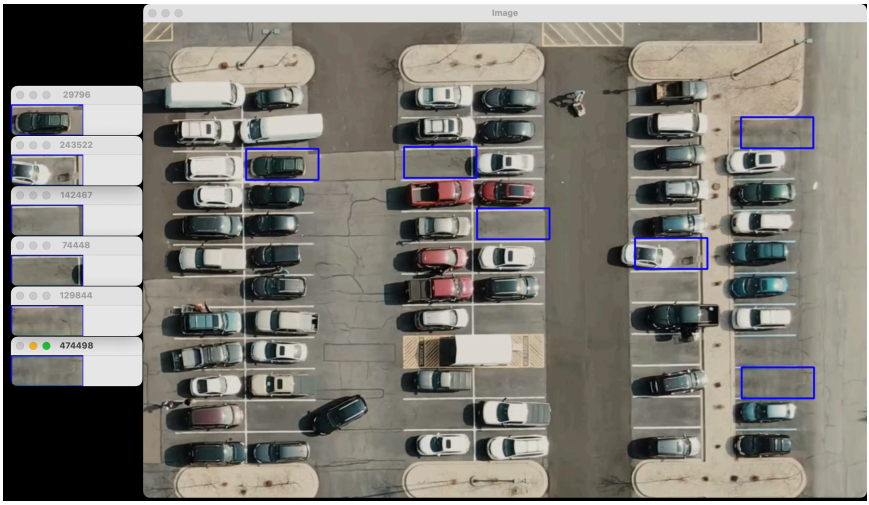
STEP 3

In this step, we need to crop all of these rectangle regions and make some operations on top of each of them so that we will detect if there is a car inside that area or not.

```
for coordinate in startingCoordinates:
    x, y = coordinate
    croppedImage = image[y:y+height, x:x+width]
    cv2.imshow(str(x*y), croppedImage)
```

Above code results in cropping and displaying of all the rectangles that we had. You can see the output below.

Since we only had 6 rectangles, there are 6 cropped little video frames that has been displayed. But when we select



all the parking lots, there will be lots of tiny video frames, that's why I will comment out the imshow() part and not display all of them each time.

STEP 4

Now, what we need to do is, we need to tell whether this region has a car present in it or not. How can I do that? I will do that by looking at its white pixel count. I need to convert those tiny region images to binary image based on its edges and corners. And from there, if they don't exceed a threshold (that will be determined later) in terms of their white pixel amount, then that lot is free; otherwise, occupied.

First, let's convert our video to grayscale video by converting each of its frames to grayscale image. How do we do that is below;

```
imageGray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
```

After that, we can blur the video frames so that some small noises will be smoothed and not going to effect the results a lot.

```
imageBlurred = cv2.GaussianBlur(imageGray, (5, 5), 1)
```

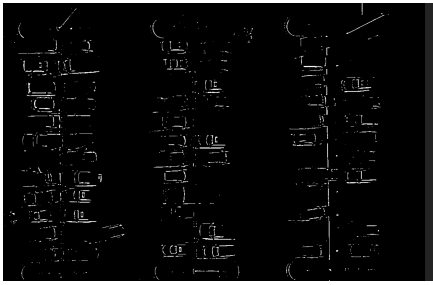
As you might have guessed, the method I used in order to smooth the frames is Gaussian Method. I used 5x5 Gaussian kernel window size and sigma value as 1. The window size and sigma values have been determined by playing and observing with different values. Resulting videos are shown below.

Left image is blurred version of the right grayscale image

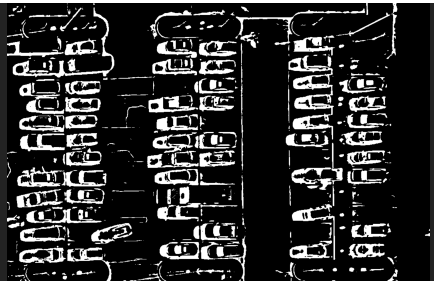
Next, we are going to use adaptiveThreshold() built-in computer vision method to convert video frames to the binary image (i.e. only black and white pixel values according to whether they exceed the threshold or not). Below, how I do that:

```
imageBinary =
cv2.adaptiveThreshold(imageBlurred, 255,
cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
cv2.THRESH_BINARY_INV, 25, 16)
```

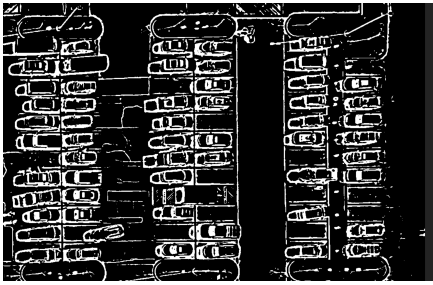
I have decided to use these values by trying several different values. Let me show some of those values' resulting images and the one that I picked.



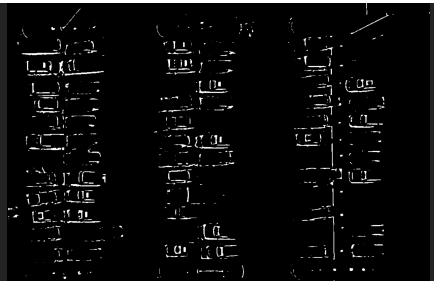
5, 16



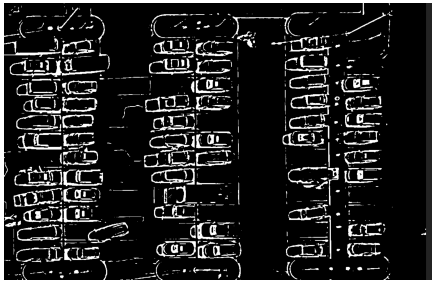
51, 16



25, 9



25,36



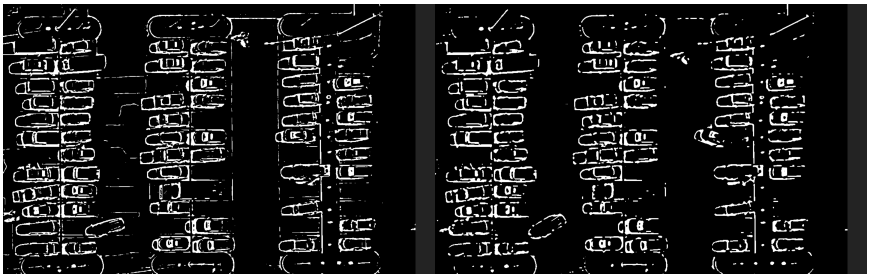
25, 16

After I have tried more than these values actually, I came up with the idea that 25,16 is the best among them.

Finally, there are two more steps that I will follow to make frames more better to distinguish the presence of cars. First, observe that there are still little bit dots here and want to get rid of them more. So I will use medianBlur() computer vision built-in method.

```
imageMedianBlur =
cv2.medianBlur(imageBinary, 5)
```

Again, the kernel size that I found useful is 5. Below, the significant change is shown.



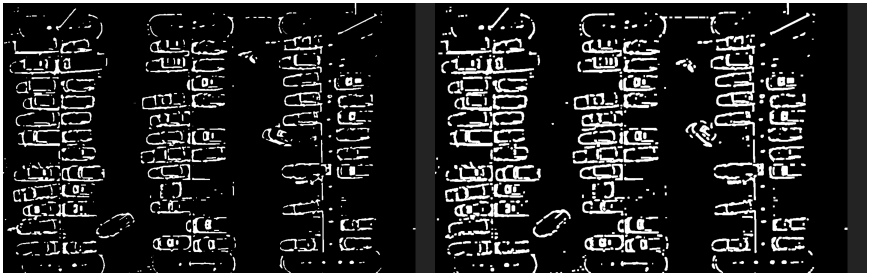
Previous one

Median blurred

Finally, let's make the white lines a bit thicker. To do that we will utilize dilate() built-in computer vision method. This time I need to input kernel as a whole, rather than just the kernel size. That's why I created the kernel first and inputted later. The code is:

```
kernel = np.ones((3, 3), np.uint8)
imageDilate = cv2.dilate(imageMedianBlur, kernel, iterations=1)
```

Again, there result of the code is shown below;



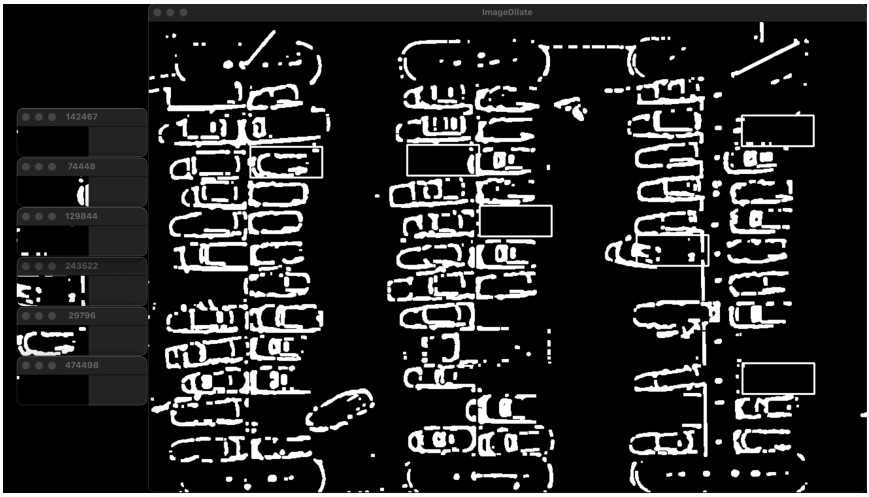
Previous One Dilated

As it is seen obviously, in the dilated one, the white pixels are more thicker than the previous one.

All of these straightforward and tiny adjustments have a little effects each time but when we compose all together, we see the overall result is way better than the first result that we had.

STEP 5

Now, we need to take this processed image and crop the rectangle areas from this one.

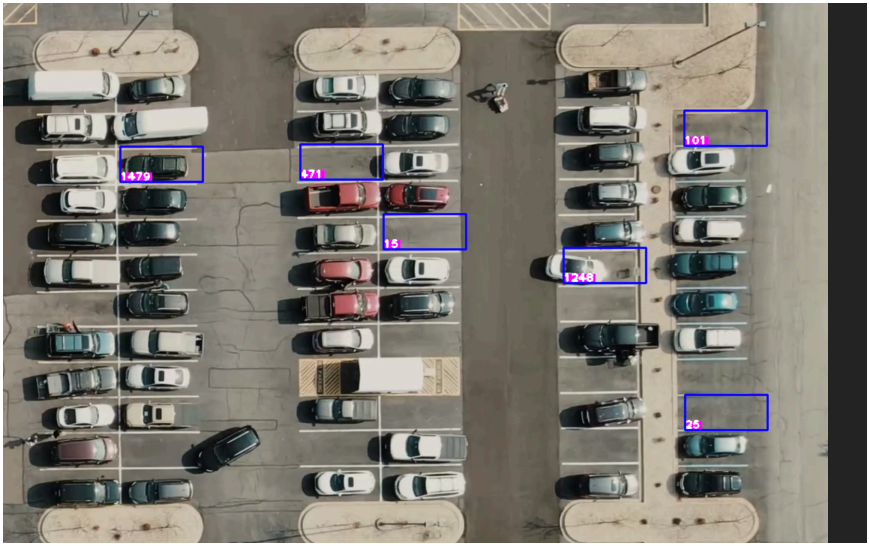


Let's count the white pixels inside each of those rectangles. If it is above the threshold that I will be assigning

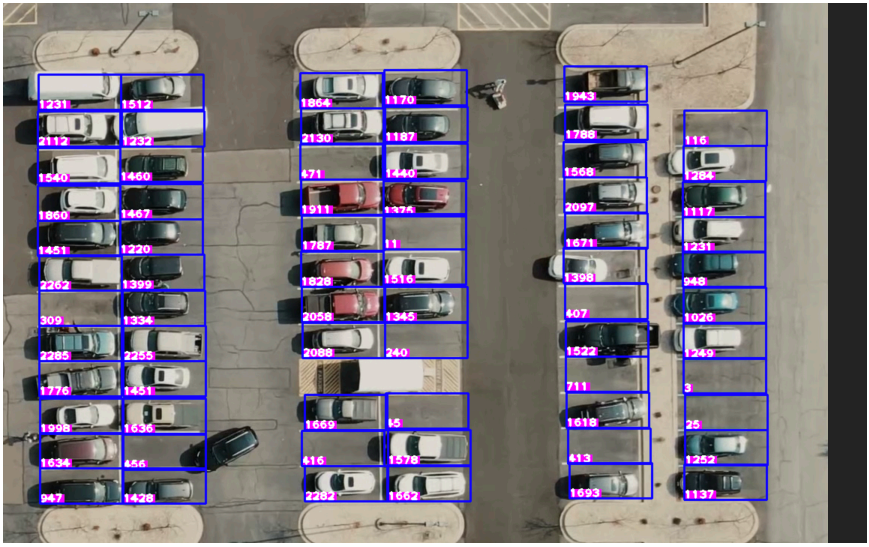
later, then it is occupied; and if it is below the threshold, then it is a free space. I will utilize countNonZero() built-in computer vision function to count the white pixels. Later, I will visualize each of their white pixel counts. How do I do this is;

```
whiteCount = cv2.countNonZero(croppedImage)
cvzone.putTextRect(image, str(whiteCount),
(x, y+height-2), offset=0, thickness=2,
scale=1)
```

And the resulting video is shown below:



As you might have noticed, the free space values vary roughly between 0 - 500, where occupied spaces are more likely to be more than 1000. However, let's not make this assumption by just looking 6 rectangles. Let's fit other remaining rectangles and observe the value variations from that video.



Notice that, the occupied zones' value vary roughly from 900 to 2300 and empty zones' from 0 to 800. To be on the safe zone, let's set the threshold for being free or occupied is 850. Thus, values above 850 are occupied, below 850 are free. Let's implement this idea.

```
whiteCount = cv2.countNonZero(croppedImage)
if whiteCount < 850:
    color = (0, 255, 0)
    thickness = 4
else:
    color = (0, 0, 0)
    thickness = 2

cv2.rectangle(image, coordinate, (coordinate[0] +
width, coordinate[1] + height), color, thickness)
```

As seen, below the threshold, I changed the rectangle color to green which indicates free parking zone; on the other hand for above the threshold, I changed the rectangle color to black which indicates zone being occupied. Below, the corresponding video screenshoot shown.

Finally, only thing that we are left of is checking how many free spaces do we have within the total amount of parking zones. So, below code calculates the free space amount and total amount and displays it live on the video.

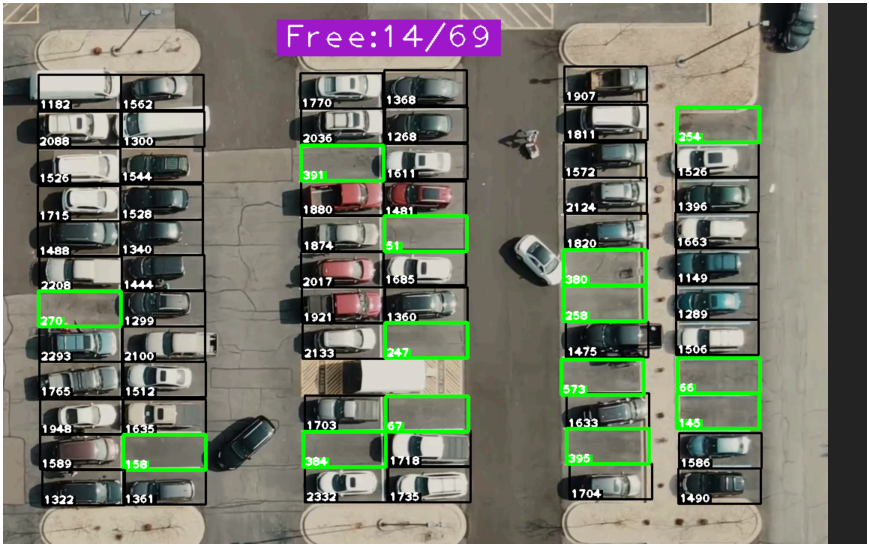
```
emptySpaceAmount = 0

if whiteCount < 850:
    color = (0, 255, 0)
    thickness = 4
    emptySpaceAmount += 1

cvzone.putTextRect(image, str(str(emptySpaceAmount)
+'/'+str(totalAmount)), (50,50), offset=0,
thickness=5, scale=5, colorR=(0, 0, 0))

cvzone.putTextRect(image, str('Free:' +
str(emptySpaceAmount) + '/' +
str(len(startingCoordinates))), (375, 60),
   offset=10, thickness=2, scale=3, colorR=(203, 26, 163))
```

First, I have declared *emptySpaceAmount* variable and initialized it do 0. Then, I added 'emptySpaceAmount += 1' line which keeps the free space amount up to date when there is a change. Finally, I have displayed the free space amount / total space amount on the live video. Finally, the resulting outcome is shown below:



Also, I have uploaded a demo video of showing the video from beginning to end. Here it is:

https://drive.google.com/file/d/1MCV7A77J9d8YpYQieKeaeX9p3R-mMcKa/view?usp=sharing

CONCLUSION

First of all, I would like to mention that, my main purpose was to take a video of any Sabanci University parking area and implement this program for that parking area. However, due to an official reason, by Directorate General of Civil Aviation of Turkey, I wasn't permitted to fly a drone on top of Sabanci University by the help of ColabSpace to film a parking area. That's why, the video and picture that I used is from an internet resource.

What I have done in this project is shortly, I have selected the parking lot and extracted them from the image which was an arbitrary frame of the video. I did this on FreeSpaceFinder.py program. Then, I have stored those parking lot coordinates information to use in main.py. In the main.py, I converted my video to grayscale, I smoothed, I extracted corners and edges, and so on by utilizing several computer vision package built-in functions. Later, I set the threshold so I can determine which space is free which is occupied by trying different values. At the end, I have visually shown free spaces and count of how many spaces within how many total amount is free.

I want to take an attention to something I believe worths talking about. I might have extract the parking lots by determining the lines beforehand and then using those spaces in order to do the same job in main.py. However, for the same reason I mentioned in the beginning of the conclusion part, I couldn't find a way to take a picture of empty parking area and then taking a video while they are filling up. I searched the internet to find both fully empty

and mainly full birdeye parking area pictures but I couldn't. Then I decided to select the parking spaces manually which in fact didn't change my main purpose (detecting the cars inside the parking spaces) of this term project.

In the future, I believe that I can extend this job I have done to broader scopes. For example, if I can have a permission to take a video of SU parking areas, I would like to make my program compatible with those areas and detect free spaces live then present it to SU car parking area users. Which is my ultimate purpose.

Thank you for this great semester, thousand thanks to people who contributed this semester so that I believe majority of the student including me had great times learning these topics. I believe that the biggest parameter of this course to be so enjoyable and valuable is the instructor staff.

Thank you,
Goktug Korkulu