FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Vision-based computation of parking lot occupancy

Authors:
Nuno Granja Fernandes up201107699
Samuel Arleo Rodriguez up201600802
Katja Hader up201602072

EIC0104 - VCOM

Abstract

This report's main concern is to devise an algorithm, using the image processing techniques acquired in the Computer Vision course, to interpret and process an image of a parking lot and identify the cars and empty slots. There may be several correct approaches to the problem and the solution will greatly depend on the set of images used.

The several approaches made to the problem included different techniques of simple morphological transformations, various types of image filtering, edge and corner detections, segmentation and clustering techniques and even image acquisition and representation.

Index Terms - Color, Hue, Perspective, Detection, Process, Pixel

Procedure

1- First Image

The first attempt to successfully detect and separate the cars present in the Parking Lot was done through a *Histogram Backprojection* followed by some morphological operations in order to better separate the cars. In order to function properly, the program must read the image of the car it wants to detect and the Parking Lot with said cars, respectively named "car.jpg" and "Park.jpg". Note that this solution does not allow much variation of car shapes and colors, since we must use as *region of interest* an image of the most generic car in the Parking Lot.

A. Histogram Backprojection

A Histogram Backprojection works by using the histogram of the *region of interest* we want to detect and a *target* image in which we want to detect it, we then compare those two images in order to build a matrix in which each pixel represents a probability of belonging to the *region of interest*, being that a higher probability is "highlighted" in whiter colors. This process can be implemented through use of *opencv's* library with the function *cv2.calcBackproject.*

We begin by computing the histogram of our *region of interest image* so we can search for it in the *target image.* After being normalized we apply a *Back Projection* in which our histogram is "laid over" our *target* image (hence *Backprojection*) and every pixel is compared in order to compute the probability of it belonging to the *region of interest*. Notice that a colored *region of interest* works much better because we have more "characteristics" to search for. Afterwards we threshold our "probability" image so that every pixel in white means that we have found our *region of interest* and we apply a logical *AND* operation between the thresholded image and the *target* image so that finally, anything that is left in the image is our *region of interest.*

After we have our *region of interest* some operations such as *erosion, dilation, threshold, opening, closing* and even some blurring algorithms like *Gaussian Blurring* were used to better segment the cars.

2- Second Image

The second approach made to identify and count the number of cars in a Parking Lot consisted of first segmenting the image and secondly applying a clustering algorithm. To run the algorithm, the code must read the images of an empty parking lot and the same parking lot with cars, named "Empty.jpg" and "Full.jpg" respectively.

A. Segmentation

To separate each car we used a pre-acquired image of the same parking lot but empty. After some image pre-processing, we proceeded to compare all 3 channels (Red, Blue, Green) of each *Full Parking Lot* pixel with the 3 channels of the corresponding pixel in the *Empty Parking Lot*. If the difference between any of the channels was bigger than a threshold defined by us, we would consider that pixel a *car pixel.* Finally, we created a new matrix in which we painted every *car pixel* in black and all other pixels in white.

To prepare our result from segmentation to the clustering algorithm we had to, again, apply some morphological operations and filtering. Then we created the matrix X in which we stored the position of every *car pixel* (x and y position), which is crucial for the clustering algorithm that follows.

B. Clustering

We opted for the *DBScan* Clustering algorithm since it is an effective and intuitive method for grouping different items in an image and it can detect any number of clusters in an image and ignore potential noise, considered *outliers* by the algorithm. The *DBScan* algorithm takes three parameters as input: Set of Points, Neighborhood and Density (*minpts*). The Set of Points will be processed and the output will be a number of clusters that obey the parameters set, Neighborhood is used to determine the limit between a similarity and other data and Density is the number of points within a Neighborhood necessary to form a cluster.

Each cluster can be represented by the position (x and y coordinates) of its *centroid*, which we stored and represented on a new matrix. The clusters we now obtained were too diverse in terms of area and shape due to the uneven shape of the cars, their angle and distance in reference to the camera, noise and reflections. One way we found to rectify this was by performing another *DBScan* Clustering, this time on the matrix containing the *centroids.* The result of applying the algorithm on a set of disperse pixels is a number of

perfectly circular clusters centred on the same pixels. One very important note is that this second Clustering must be performed with a Density (*minpts*) parameter of 1, otherwise the Clustering wouldn't be done since each pixel (*centroid*) is alone in its neighborhood.

We are now left with *n* perfectly circular clusters on top of each car we were able to detect. Ideally *n* would be equal to 1 so the number of clusters would be equal to the number of cars in the Parking Lot.

Conclusion

Segmentation and Clustering algorithms proved to be a very intelligent and interesting approach to the problem keeping in mind that it must be followed by an algorithm that successfully counts the number of segments/clusters. This experience proved very effective in familiarizing us with many image processing methods and also raising awareness to the difficulty that arises with obstacles like noise, reflections, different perspectives in image acquisition and irregularities of the image to be processed.

Annexes

First Approach

```
import numpy as np
import cv2
from matplotlib import pyplot as plt

def printImage(image):
cv2.imshow("window",image)
cv2.waitKey(0)
cv2.destroyAllWindows()

# Region of Interest in HSV
roi = cv2.imread('car.jpg')
hsv = cv2.cvtColor(roi,cv2.COLOR_BGR2HSV)

#Target is the image we search the car in
target = cv2.imread('Park.jpg')
hsvt = cv2.cvtColor(target,cv2.COLOR_BGR2HSV)

# calculating object histogram
roihist = cv2.calcHist([hsv],[0, 1], None, [180, 256], [0,
180, 0, 256] )

# normalize histogram and apply backprojection
# Convolute with a circular disc
kernel =
cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(5,5))
cv2.filter2D(dst,-1,kernel,dst)

# Applying a Threshold to the 'Probability' matrix
ret,thresh = cv2.threshold(dst,50,255,0)
thresh = cv2.merge((thresh,thresh,thresh))

# Applying a Logical AND operation to maintain only the
region of interest
res = cv2.bitwise_and(target,thresh)
cv2.imwrite('res.jpg',res)
printImage(res)

# Kernel created for the following morphological
operations
kernel = np.ones((3,3),np.uint8)

# Erosion on resulting image
erosion = cv2.erode(res,kernel,iterations = 3)
printImage(erosion)
```

```
cv2.normalize(roihist,roihist,0,255,cv2.NORM_MINMAX
)
dst =
cv2.calcBackProject([hsvt],[0,1],roihist,[0,180,0,256],1)

# Painting every white pixel black
dilation[dilation>180] = 1
printImage(dilation)

#Threshold
ret, thresh =
cv2.threshold(dilation,100,255,cv2.THRESH_BINARY)
printImage(thresh)

# Second Erosion on threshold image
erosion2 = cv2.erode(thresh,kernel,iterations = 5)
printImage(erosion2)

# Second Dilation on Eroded image
dilation2 = cv2.dilate(erosion2,kernel,iterations = 5)
printImage(dilation2)
```

```
# Dilation on Resulting image
dilation = cv2.dilate(erosion,kernel,iterations = 11)
printImage(dilation)
```

Second Approach

```
import numpy as np
import cv2
from matplotlib import pyplot as plot
from sklearn.cluster import DBSCAN
import random as rd
import os
import sys

def numImages(list):
	tam = len(list)
	i = 0
	while list[i] is not None:
		i+=1
		if i == tam:
			break
	return i

# MatPlotLib method for printing images
def printImage(img1,img2=None,img3=None,img4=None):
	images = [img1,img2,img3,img4]
	n = numImages(images)
	for i in range(0,n):
		if n<3:
			plot.subplot(1,n,i+1),plot.imshow(images[i],cmap='Greys_r')
			plot.title(str(i)), plot.xticks([]), plot.yticks([])
		else:
			plot.subplot(2,2,i+1),plot.imshow(images[i],cmap='Greys_r')
			plot.title(str(i)), plot.xticks([]), plot.yticks([])
	plot.show()

# OpenCV method for printing images
def printImage(img):
	# Press ESC to exit
	if img is not None:
		cv2.imshow('res',img)
		cv2.waitKey(0)
		cv2.destroyAllWindows()
	else:
		print "Problem printing image."

def getChannels(img):
	return img[:,:,2],img[:,:,1],img[:,:,0]

def applyThreshold(gray,min_thr,max_thr):
	rows,col = gray.shape
	thr = np.zeros((rows,col)) # Creating a null matrix for copying matrix gray2
	np.copyto(thr,gray)
	for i in range(0,rows):
		for j in range(0,col):
```

```
                        if min_thr <= thr[i,j] and thr[i,j] <= max_thr:
                                thr[i,j] = 0
                        else:
                                thr[i,j] = 255
        return thr

def thresholdImage(gray,thr_type,thr,block_size=None,img=None):

        """ Where thr_type in {1,2,3,4}
                1: Normal threshold
                2: Otsu
                3: Adaptive (mean)
                4: Adaptive (Gaussian)
                More thresholds: Using two thresholds taking into account that most pixels are from the floor
                        (Trying to don't erase black cars)
                5: Double threshold (using percentiles)
                6: Double threshold (using manually set values)
        """
        if thr_type == 1:
                ret,thr = cv2.threshold(gray,thr,255,cv2.THRESH_BINARY)
                return thr
        elif thr_type == 2:
                ret,thr = cv2.threshold(gray,0,255,cv2.THRESH_BINARY+cv2.THRESH_OTSU) # Black/red cars
desapeared. Good for Segmentation of background
                return thr
        elif thr_type == 3:
                return
cv2.adaptiveThreshold(gray,255,cv2.ADAPTIVE_THRESH_MEAN_C,cv2.THRESH_BINARY,block_size,2) # Less
noise, but can't recognize all cars
        elif thr_type == 4:
                return
cv2.adaptiveThreshold(gray,255,cv2.ADAPTIVE_THRESH_GAUSSIAN_C,cv2.THRESH_BINARY,block_size,2) # More
noise, more cars
        elif thr_type == 5:
                firstQ = np.percentile(gray2,65) # Return de value of the pixel that corresponds to the 65 percent of all
sorted pixels in grayscale
                secondQ = np.percentile(gray2,50)
                thirdQ = np.percentile(gray2,35)
                return applyThreshold(gray,firstQ,thirdQ)
        elif thr_type == 6:
                return applyThreshold(gray,40,113)
        elif thr_type == 7:
                rows,col = img[:,:,0].shape
                r1,g1,b1 = getChannels(gray) # Actually is not grayscale but a BGR image (just a name)
                r2,g2,b2 = getChannels(img)
                res = np.zeros((rows,col))
                for i in range(0,rows):
                        for j in range(0,col):
                                rDif = abs(int(r1[i,j]) - int(r2[i,j]))
                                gDif = abs(int(g1[i,j]) - int(g2[i,j]))
                                bDif = abs(int(b1[i,j]) - int(b2[i,j]))
                                if rDif >= thr or gDif >= thr or bDif >= thr:
                                        res[i,j] = 0
                                else:
```

```
                                        res[i,j] = 255
                return res

        else:
                return None

def getEdges(gray,detector,min_thr=None,max_thr=None):
        """
                Where detector in {1,2,3,4}
                1: Laplacian
                2: Sobelx
                3: Sobely
                4: Canny
                5: Sobelx with possitive and negative slope (in 2 negative slopes are lost)
        """
        if min_thr is None:
                min_thr = 100
                max_thr = 200
        if detector == 1:
                return cv2.Laplacian(gray,cv2.CV_64F)
        elif detector == 2:
                return cv2.Sobel(gray,cv2.CV_64F,1,0,ksize=-1)
        elif detector == 3:
                return cv2.Sobel(gray,cv2.CV_64F,0,1,ksize=-1)
        elif detector == 4:
                return cv2.Canny(gray,min_thr,max_thr)  # Canny(min_thresh,max_thresh) (threshold not to the
intensity but to the
                                                                                                        #
intensity gradient -value that measures how different is a pixel to its neighbors-)
        elif detector == 5:
                sobelx64f = cv2.Sobel(gray,cv2.CV_64F,1,0,ksize=5)
                abs_sobel64f = np.absolute(sobelx64f)
                return np.uint8(abs_sobel64f)

def dbscan(points,eps,min_samples):
        db = DBSCAN(eps=eps, min_samples=min_samples).fit(points) # eps=5 min_samples = 80

        # Labeling pixels by cluster
        core_samples_mask = np.zeros_like(db.labels_, dtype=bool)
        core_samples_mask[db.core_sample_indices_] = True
        labels = db.labels_

        # Number of clusters in labels, ignoring noise if present.
        n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)

        # Creating list of clusters
        return [points[labels == i] for i in xrange(n_clusters_)]

def getCentroids(clusters,img):
        n = 0
        # centroid will store the coordinates of the center of each cluster
        centroids = np.zeros((len(clusters),2),dtype=np.int_)
        for c in clusters:
                x,y = (int(sum(c[:,0])/len(c[:,0])),int(sum(c[:,1])/len(c[:,1])))
```

```
                    r,g,b = rd.randint(0,255),rd.randint(0,255),rd.randint(0,255)
                    centroids[n,0],centroids[n,1] = x,y
                    n = n + 1
                    cv2.circle(img,(y,x),7,(r,g,b),-1)
        return centroids,img

def paintClusters(img,clusters):
        # Painting clusters
        for c in clusters:
                r,g,b = rd.randint(0,255),rd.randint(0,255),rd.randint(0,255)
                for pixel in c:
                        img[int(pixel[0]),int(pixel[1]),:] = b,g,r
        return img


# Empty and non-empty parking lot image
img1 = cv2.imread("Empty.jpg")
img2 = cv2.imread("Full.jpg")
if img1 is None or img2 is None:
        print("It was not possible to load the images. Please check paths.")
        sys.exit()

# Dimensions of the image
rows,col = img1[:,:,0].shape

# Adapting images sizes
img2[:,:,0] = img2[:rows,:col,0]
img2[:,:,1] = img2[:rows,:col,1]
img2[:,:,2] = img2[:rows,:col,2]

# Blurring empty parking lot image
blur = cv2.GaussianBlur(img1,(7,7),0)

# Images for storing results
res2 = img2.copy()
res3 = img2.copy()
res4 = img2.copy()
res5 = img2.copy()

# Auxiliar image for storing results
res = np.zeros((rows,col))

# Gray scale image of the non-empty parking lot
gray = cv2.cvtColor(img2,cv2.COLOR_BGR2GRAY)

# Edges on non-empty parking lot using Canny edge detector
edges = getEdges(gray,4,50,200)

# Threshold for every channel
threshold = 40

# Applying color threshold: It takes each channel of the empty parking lot image
# and compares each pixel with the same channel on the non-empty park. If this difference
# is more than the threshold, then the pixel is considered as a car pixel and the pixel
```

```
# on the same position in the binary image "res" is painted black. This allows to recognize
# cars of almost any color, however cars with intisities similar to the background are
# hard to paint.
res = thresholdImage(img2,7,threshold,img=blur)

# Adding edges to the thresholded image to separate cars
res = res+edges

# Getting indexes of car pixels (black pixels) on res
cars_pixels = np.where(res == 0)

# Number of car pixels
num_pix = cars_pixels[0].size

# Declaring matrix with the same number of rows as black pixels the image has
X = np.zeros((num_pix,2))

# Putting coordinates of pixels in X
for i in range(0,num_pix):
        X[i,:] = [cars_pixels[0][i],cars_pixels[1][i]]

# Applying dbscan clustering to X (pixels positions) where:
# eps: distance from current centroid to others
# min_samples: min number of pixels that must be at a distance of less or equal than eps
# to consider the current pixel as a centroid
eps = 5
min_samples = 80

clusters = dbscan(X,eps,min_samples)

# Centroids stores the coordinates of the center of each cluster
centroids,res2 = getCentroids(clusters,res2)

if centroids.size == 0:
        print("No clusters were created. Please change dbscan parameters.")
        sys.exit()

# Trying to cluster the centroids
eps2 = 30
min_samples2 = 1

# Creating list of clusters
clusters2 = dbscan(centroids,eps2,min_samples2)
#clusters2 = (clusters2[:,1],clusters2[:,0])

# centroid will store the coordinates of the center of each cluster
centroids2,res3 = getCentroids(clusters2,res3)

# Trying to cluster the centroids again
eps3 = 45
min_samples3 = 1

# Creating list of clusters
clusters3 = dbscan(centroids2,eps3,min_samples3)
```

```
#clusters2 = (clusters2[:,1],clusters2[:,0])

# centroid will store the coordinates of the center of each cluster
centroids3,res5 = getCentroids(clusters3,res5)

# Painting clusters
res4 = paintClusters(res4,clusters)

printImage(res)
printImage(res2)
printImage(res3)
printImage(res5)
printImage(res4)

# Saving resulting image
cv2.imwrite("PRUEBAres_thr_"+str(threshold)+"_eps1_"+str(eps)+"_min1_"+str(min_samples)+
                                                    "_eps2_"+str(eps2)+"_min2_"+str(min_samples2)+".jpg",res3)


# If you want to calculate an approximate number of empty spaces write the number of
# available spaces on the empty parking lot picture in the following variable:
"""
capacity = 76
num_cars = len(clusters3)
if num_cars >= capacity :
        print("There is no place where to park.")
else:
        print("There are available spaces.")
"""

print(len(centroids2))
print(len(centroids3))
```