



**Wydział Elektroniki
i Technik Informacyjnych**

POLITECHNIKA WARSZAWSKA

Przetwarzanie cyfrowe obrazów

Semestr Zimowy 2022

RAPORT

Projekt:

Detekcja loga firmy McDonald's w obrazach

Prowadzący:

dr hab. inż. Tomasz Trzciński

Wykonawca:

Mikołaj Bochiński - 300484

Warszawa, 22 stycznia 2023

Cel projektu:

Dla indywidualnie wybranej klasy obrazów dobrać, zaimplementować i przetestować odpowiednie procedury wstępного przetworzenia, segmentacji, wyznaczania cech oraz identyfikacji obrazów cyfrowych. Powstały w wyniku projektu program powinien poprawnie rozpoznawać wybrane obiekty dla reprezentatywnego zestawu obrazów wejściowych.

Interesujący nas obiekt - żółte „M”.

Zbiór obrazów testowych wybranych do projektu:







Opis działania algorytmu:

- Wstępne przetworzenie obrazu

Po wczytaniu pliku, następuje jego przeskalowanie. Jest to niezbędny krok z uwagi na duży rozmiar zdjęć (rozdzielcość 4k). Ich interpolacja do rozmiaru 1200, 900 znacznie skraca czas obliczeń.

W kolejnym kroku następuje konwersja zdjęcia z domyślnej przestrzeni RGB na format YUV. Ma on tę zaletę, że posiada wydzielony kanał luminancji (Y).

```
def convertBGR2YUV(image):
    image_YUV = np.zeros(image.shape, dtype='uint8')

    YUV_lut = np.array([[0.114, 0.587, 0.299], [0.436, -0.289, -0.147], [-0.100, -0.515, 0.615]])

    for row in range(image.shape[0]):
        for col in range(image.shape[1]):
            B = image[row][col][0]
            R = image[row][col][2]
            Y = cap_value_to_8bit(np.matmul(YUV_lut[0], image[row][col]))
            U = cap_value_to_8bit((B - Y) * 0.492 + 128)
            V = cap_value_to_8bit((R - Y) * 0.877 + 128)

            image_YUV[row][col] = (np.int([Y, U, V])).astype(int)

    return image_YUV
```

Metoda służąca do konwersji z BGR na YUV



Zdjęcie przekonwertowane do YUV

Następnie kanał Y zostaje poddany operacji wyrównania histogramu. Dzięki temu zabiegowi kontury obiektów mniej zlewają się z tłem, co ułatwia późniejszą segmentację. Efekt operacji przedstawiono poniżej.

```
def equalize_histogram(image_channel):
    equalized_image = np.zeros(image_channel.shape, dtype='uint8')

    num_of_pixels = image_channel.shape[0] * image_channel.shape[1]
    num_of_pixels_for_value = np.zeros(256, dtype=int)
    equalizing_LUT = np.zeros(256, dtype=int)

    for row in range(image_channel.shape[0]):
        for col in range(image_channel.shape[1]):
            pixel_value = image_channel[row][col]
            num_of_pixels_for_value[pixel_value] += 1

    probabilites_sum = 0
    for i in range(256):
        probabilites_sum += num_of_pixels_for_value[i]
        equalizing_LUT[i] = (np.rint(probabilites_sum*255/num_of_pixels)).astype(int)

    for row in range(image_channel.shape[0]):
        for col in range(image_channel.shape[1]):
            equalized_image[row][col] = equalizing_LUT[image_channel[row][col]]

    return equalized_image
```

Kod operacji wyrównania histogramu



Obraz po wyrównaniu histogramu składowej Y

Później obraz zamieniany jest z powrotem do przestrzeni BGR, za pomocą zaimplementowanej funkcji.

```
def convertYUV2BGR(image):
    image_BGR = np.zeros(image.shape, dtype='uint8')

    BGR_lut = np.array([[1, 2.03211, 0], [1, -0.39465, -0.58060], [1, 0, 1.13983]])

    for row in range(image.shape[0]):
        for col in range(image.shape[1]):
            Y = image[row][col][0]
            U = image[row][col][1]
            V = image[row][col][2]

            B = cap_value_to_8bit(np.matmul(BGR_lut[0], [Y, U - 128, V - 128]))
            G = cap_value_to_8bit(np.matmul(BGR_lut[1], [Y, U - 128, V - 128]))
            R = cap_value_to_8bit(np.matmul(BGR_lut[2], [Y, U - 128, V - 128]))
            image_BGR[row][col] = (np.int([B, G, R])).astype(int)

    return image_BGR
```

Metoda służąca do konwersji z YUV na BGR

- Progowanie i binaryzacja

W kolejnym kroku dokonano operacji konwersji otrzymanego obrazu do przestrzeni HLS.

```
def convertBGR2HLS(image):
    image_HLS = np.zeros(image.shape, dtype='uint8')

    for row in range(image.shape[0]):
        for col in range(image.shape[1]):
            B = image[row][col][0] / 255
            G = image[row][col][1] / 255
            R = image[row][col][2] / 255

            V_max = max(B, G, R)
            V_min = min(B, G, R)
            L = (V_max + V_min)/2
            S = 0
            if L < 0.5:
                S = (V_max-V_min)/(V_max+V_min)
            else:
                S = (V_max-V_min) / (2-(V_max+V_min))

            V_diff = V_max-V_min
            H = 0
            if V_max == R:
                H = 60*(G-B)/V_diff
            elif V_max == G:
                H = 120 + (60*(B-R))/V_diff
            elif V_max == B:
                H = 240 + (60*(R-G))/V_diff
            elif R == G and G == B:
                H = 0

            if H < 0:
                H = H + 360

            L = cap_value_to_8bit(255*L)
            S = cap_value_to_8bit(255*S)
            H = H / 2
            image_HLS[row][col] = (np.int([H, L, S])).astype(int)

    hls_cv = cv2.cvtColor(image, cv2.COLOR_BGR2HLS)
    return image_HLS
```

Kod algorytmu konwersji BGR na HLS

```

def convertHLS2BGR(image):
    image_BGR = np.zeros(image.shape, dtype='uint8')

    for row in range(image.shape[0]):
        for col in range(image.shape[1]):
            H = image[row][col][0]
            L = image[row][col][1] / 255
            S = image[row][col][2] / 255

            C = (1-abs(2*L-1))*S
            H_prim = H / 30 # we divide by 30 not 60 because hue is stored as 0-180 value, not 0-360
            X = C * (1-abs(H_prim % 2 - 1))

            R = G = B = 0
            if 0 <= H_prim < 1:
                R = C
                G = X
            elif 1 <= H_prim < 2:
                R = X
                G = C
            elif 2 <= H_prim < 3:
                G = C
                B = X
            elif 3 <= H_prim < 4:
                G = X
                B = C
            elif 4 <= H_prim < 5:
                R = X
                B = C
            elif 5 <= H_prim < 6:
                R = C
                B = X

            m = L - C/2
            image_BGR[row][col] = (np.array([(B+m)*255, (G+m)*255, (R+m)*255])).astype(int)

    bgr_cv = cv2.cvtColor(image, cv2.COLOR_HLS2BGR)
    return image_BGR

```

Kod algorytmu konwersji HLS na BGR

Format HLS wygodnie wykorzystuje się podczas procesu progowania. Polega on na wydzieleniu interesujących nas pixeli należących do obiektów i pixeli tła. Odbywa się to poprzez sprawdzenie, czy wartość HLS dla danego pixela zawiera się w zdefiniowanym przedziale:

```

MIN_VALUE = [16, 40, 150]
MAX_VALUE = [30, 240, 255]

```

Wartości min i max opisują kolory graniczne jakie może przyjmować szukany przez nas obiekt - żółte „M” - w zależności od warunków.

Dla każdego pixela w obrazie wywoływana jest metoda `check_if_color_in_threshold()`.

```

def check_if_color_in_threshold(pixel):
    MIN_VALUE = [16, 40, 150]
    MAX_VALUE = [30, 240, 255]

    if pixel[0] >= MIN_VALUE[0] and pixel[0] <= MAX_VALUE[0] and \
       pixel[1] >= MIN_VALUE[1] and pixel[1] <= MAX_VALUE[1] and \
       pixel[2] >= MIN_VALUE[2] and pixel[2] <= MAX_VALUE[2]:
        return True
    else:
        return False

```

Metoda odpowiadająca za progowanie

Jeżeli wynikiem operacji porównania jest True, wartość pixela ustawiana jest na [255, 255, 255] - kolor biały. W przeciwnym przypadku [0, 0, 0] - czarny.



Uzyskany obraz binarny

Ponieważ obiekty w niektórych miejscach są nieciągłe:



mogą wystąpić problemy przy późniejszym rozpoznawaniu.

W celu poprawy „wypełnienia” i spójności obiektów stosowana jest operacja zamknięcia: dylacja, a następnie erozja. Są to dwa splotowe filtry rankingowe, które jako argumenty przyjmują obraz binarny oraz dowolny nieparzysty rozmiar maski splotu. Spośród wszystkich wartości znajdujących się w buforze dla danego pixela wybierana jest wartość najmniejsza lub największa, a następnie ustawiana jako jego wartość.

Wadą zaimplementowanych filtrów jest długi czas działania. Mają złożoność obliczeniową $O(n^4)$.

```

def dilation(binary_image, mask_size):
    assert mask_size % 2 == 1, f"Odd number expected, got: {mask_size}"
    image_after_dilation = binary_image.copy()

    offset = math.floor(mask_size/2)
    #first and last {offset} rows/columns are skipped
    for row in range(offset, binary_image.shape[0] - offset):
        for col in range(offset, binary_image.shape[1] - offset):
            pixel_neighbourhood = np.zeros((mask_size, mask_size, 3))

            for r in range(-offset, offset+1):
                for c in range(-offset, offset+1):
                    pixel_neighbourhood[r+1][c+1] = binary_image[row+r][col+c]

            image_after_dilation[row][col] = np.amax(pixel_neighbourhood)

    return image_after_dilation

```

Kod filtra dylacji

```

def erosion(binary_image, mask_size):
    assert mask_size % 2 == 1, f"Odd number expected, got: {mask_size}"
    image_after_erosion = binary_image.copy()

    offset = math.floor(mask_size/2)
    #first and last {offset} rows/columns are skipped
    for row in range(offset, binary_image.shape[0] - offset):
        for col in range(offset, binary_image.shape[1] - offset):
            pixel_neighbourhood = np.zeros((mask_size, mask_size, 3))

            for r in range(-offset, offset+1):
                for c in range(-offset, offset+1):
                    pixel_neighbourhood[r+1][c+1] = binary_image[row+r][col+c]

            image_after_erosion[row][col] = np.amin(pixel_neighbourhood)

    return image_after_erosion

```

Kod filtra erozji

- Segmentacja

Obraz po korekcji poddawany jest następnie procesowi segmentacji. Tworzona jest lista przechowująca obiekty klasy Segment, zawierające informacje o parametrach obiektu (pixele, rozmiar itp. {ze względu na dużą objętość, klasa dostępna do analizy w kodzie źródłowym}). Do zgrupowania białych pikseli w obiekty wykorzystano algorytm *Flood fill*.

Porównuje on wartości pikela oraz pikeli sąsiednich (lewo, prawo, góra, dół). Jeżeli są równe, sąsiedzi trafiają do kolejki. Po zgrupowaniu, segmentowi nadawany jest losowy kolor (w celu wizualizacji).

```

def flood_fill(pixel, segment_color, image, segments_list):
    colored_image = image.copy()
    segment_pixels = []
    pixels_queue = []
    pixels_queue.append(pixel)

    while len(pixels_queue) > 0:
        current_pixel = pixels_queue[0]
        pixels_queue.pop(0)

        colored_image[current_pixel][:] = segment_color
        segment_pixels.append(current_pixel)

        if current_pixel[0] != 0 and current_pixel[0] != image.shape[0]-1 and current_pixel[1] != 0 and current_pixel[1] != image.shape[1]-1:
            left_pixel = (current_pixel[0], current_pixel[1]-1)
            right_pixel = (current_pixel[0], current_pixel[1]+1)
            top_pixel = (current_pixel[0]-1, current_pixel[1])
            bottom_pixel = (current_pixel[0]+1, current_pixel[1])

            if left_pixel not in pixels_queue and is_pixel_white(colored_image[left_pixel]):
                pixels_queue.append(left_pixel)

            if right_pixel not in pixels_queue and is_pixel_white(colored_image[right_pixel]):
                pixels_queue.append(right_pixel)

            if top_pixel not in pixels_queue and is_pixel_white(colored_image[top_pixel]):
                pixels_queue.append(top_pixel)

            if bottom_pixel not in pixels_queue and is_pixel_white(colored_image[bottom_pixel]):
                pixels_queue.append(bottom_pixel)

        segments_list.append(Segment(segment_color, segment_pixels, colored_image))

    return colored_image

```

Kod algorytmu `flood_fill`



Obraz podzielony na segmenty

Z uwagi na znaczną liczbę niewielkich segmentów tworzących „szum”, obiekty filtrowane są po rozmiarze (ilości pikeli, które do niego należą). W przypadku segmentów o polu mniejszym niż 300 pikeli usuwa się je z listy. Przyspieszy to późniejszą analizę.

```
def filter_substantial_segments(segments, image):
    for segment in reversed(segments):
        if segment.calculate_area() < 300:
            for pixel in segment.pixels:
                image[pixel][0] = 0
                image[pixel][1] = 0
                image[pixel][2] = 0

    segments.remove(segment)
```

Funkcja filtrująca segmenty



Obraz po redukcji niewielkich segmentów

- **Identyfikacja**

Podczas tworzenia obiektów klasy Segment obliczane są współczynniki i niezmienniki:

- W3, W7
- M1, M2, M3, M7

Dla loga firmy wyznaczono powyższe wartości. Pozwalają one na, określenie czy segment zaliczyć jako logo.



Wyznaczone nierówności:

1. na podstawie logo

```
if (segment.M1 >= 0.4 and segment.M1 <= 0.6 and  
    segment.M2 >= 0.005 and segment.M2 <= 0.06 and  
    segment.M7 >= 0.018 and segment.M7 <= 0.1 and  
    segment.W3 >= 1.8 and segment.W3 <= 4 and  
    segment.W7 >= 0.75 and segment.W7 <= 1) or \
```

2. niektóre mniejsze logo lub mocno obrócone, sprawiały trudność przy wykrywaniu, stąd rozszerzono zakres każdego ze współczynników, dodano jednak ograniczenie na stosunek długości do szerokości segmentu (szukane logo jest prawie zawsze kwadratowe)

```
(width/height < 1.33 and height/width < 1.33 and  
0.1 <= segment.M1 <= 1 and  
0.00001 <= segment.M2 <= 0.3 and #0.23  
0.01 <= segment.M7 <= 0.2 and  
1 <= segment.W3 <= 4 and  
0.75 <= segment.W7 <= 1):
```

Zakwalifikowane segmenty oznaczane są na początkowym obrazie.

```
def mark_logos_on_image(segments, image):
    image_with_logos = image.copy()
    for segment in segments:
        image_with_logos = segment.mark_logo(image_with_logos)

    return image_with_logos
```

Oznaczenie wszystkich logo ramką

Wyniki działania algorytmu:







Wnioski

Największą wadą algorytmu jest czas jego działania. Dla zdjęć o rozmiarze 1200x900, średni czas obliczeń to 90s. Jest to spowodowane przede wszystkim znaczną liczbą metod, która przechodzi wielokrotnie po obrazie z użyciem zagnieżdżonych pętli. Sama operacja zamknięcia (dylacja i erozja), z użyciem najmniejszej maski 3 zajmuje średnio 30 s. Dodatkowo zaimplementowane przeze mnie metody służące, np. konwersji między przestrzeniami barw stawiają przede wszystkim na czytelność, a nie na optymalizację czasu działania. Metody wbudowane w openCV, czy numpy są praktycznie zawsze o wiele wydajniejsze.

Ze względu na rozszerzony warunek detekcji (druga zależność) algorytm wykrywa niekiedy fałszywe obiekty. Przykład przedstawiono poniżej. W tym przypadku może być to jednak związane z niską rozdzielczością zdjęcia.

Sądzę jednak, że lepiej aby jak najwięcej logo było wykrywanych nawet jeśli od czasu do czasu detekcja zadziała niepoprawnie.

