به نام خدا

گزارش پروژه فشرده‌سازی تصویر

نام دانشجو

امیر خسروی نژاد

شماره دانشجویی

۹۸۳۱۱۱۳

سوال اول:

مشکل این است که اگر مثلا فاکتور Cb و Cr یا به طور کلی شدت رنگ پیکسلهای کناری و بلاکهای مجاور بالاتر باشد، تغییر رنگ بسیار زیادی را احساس می‌کنیم و خوشایند نیست. البته این مورد هم که روشنایی خیلی زیاد باشد، تاثیر زیادی دارد چون چشم ما به روشنایی حساس‌تر است.

سوال دوم:

هر چقدر سایز بلاک بیشتر شود، نرخ فشرده‌سازی بالاتر می‌رود ولی خب کیفیت عکس را از دست خواهیم داد. در مقابل، هر چه سایز بلاک هم کاهش یابد،

سوال سوم:

before compression: 120

8 -> 00

34 -> 01

5 -> 1000

6 -> 1001

43 -> 1010

127 -> 1011

10 -> 11

after compression: 38

سوال چهارم:

حجم فایل درخت هافمن مربوط به ضرایب AC حدود ۱۸۰ بایت و ضرایب DC حدود ۲ مگابایت شد!

توضیح مختصر راجع به کد:

```python
image_ = im.open('photo1.png')
#image_.show()
width = image_.size[0]
height = image_.size[1]
#print(width, height)
#YCbCr_withoutchroma_subsampling(image_)
for i in range(width):
    for j in range(height):
        data = image_.getpixel((i,j))
        #print(data)
        #print(data) #(255, 255, 255)
        r = data[0]
        g = data[1]
        b = data[2]
        Y = 16 + (65.738 * r / 256) + (109.057 * g / 256) + (25.064 * b / 256)
        if (i % 2 == 0 and j % 2 == 0):
            Cb = 128 - (37.945 * r / 256) - (74.494 * g / 256) + (112.439 * b / 256)
            Cr = 128 + (112.439 * r / 256) - (95.154 * g / 256) - (18.285 * b / 256)
        else:
            Cb = image_.getpixel((i - (i % 2), j - (j % 2)))[1]
            Cr = image_.getpixel((i - (i % 2), j - (j % 2)))[2]
        image_.putpixel((i,j), (int(Y), int(Cb), int(Cr)))
#image_.show()
#image_.save('with_sub.png')
```

همانطور که در تصویر فوق مشخص است، ابتدا عکس اولیه را باز میکنیم. رنگهای هر پیکسل در متغیرهای r, g, b ریخته میشود و طبق فرمول زیر از سیستم rgb به YCbCr میرود. اما نکته‌ای باید در این تکه کد توجه داشت این است که با subsample 4:2:0 صورت گرفته یعنی به طور جزئی در هر ۲*۲ در پیکسلها، مقدار Y قبل و بعد از subsampling ثابت و برای Cb, Cr از خانه بالا سمت چپ روی ۳ خانه دیگر کپی میشود که پیاده‌سازی عملی آن را در کد می‌بینیم. (در فایلها ۲ عکس با فرمت تصویر YCbCr قبل و بعد از subsampling موجود است)

```python
main.py > block_8x8

def block_8x8(width, height):
    dcts = [[0.0] * height] * width
    for i in range(0, width, 8):
        for j in range(0, height, 8):
            arr = [[0] * 8 ] * 8
            for b_x in range(i, i + 8):
                for b_y in range(j, j + 8):
                    if (b_x >= width) or (b_y >= height):
                        data = (16, 128, 128)
                    else:
                        data = image_.getpixel((b_x, b_y))
                    arr[b_x - i][b_y - j] = data
            #if i == 0 and j == 0:
                #print(arr)
            #dct_out_Y = dct_2D(arr, 0)
            #dct_out_Cb = dct_2D(arr, 1)
            #dct_out_Cr = dct_2D(arr, 2)
            #print(dct_out_Y)
            """list_asli = []
            for s1 in range(8):
                lis_ = []
                for s2 in range(8):
                    lis_.append((dct_out_Y[s1][s2], dct_out_Cb[s1][s2], dct_out_Cr[s1][s2]))
                list_asli.append(lis_)"""
            dcts[i][j] = dct_2D(arr)
            #print("i = {}, j = {}".format(i, j))
            #print(dcts[i][j])
            #if i == 400 and j == 80:
                #print(i, j, dcts[i][j][0][0][0])
                #for k in range(8):
                    #for l in range(8):
                        #print(i, j, dcts[i][j][k][l][0])
            #for item in arr:
                #print(item)
    return dcts
```

تابع بعدی در بالا مشخص است: این تابع پیمایش ۸ واحد ۸ واحد روی پیکسلها، بلاکهای ۸ تایی را در لیست ۲بعدی (arr) میریزد و این مورد که اگر بلاک ناقص ماند، با مشکی پر شود در خط ۵۸ نوشته شده است. بعد از اینکه یک بلوک ۸ تایی ساخته شد به تابع dct_2D داده که با استفاده ار کتابخانه Scipy این تبدیل انجام شده است. در نهایت، بلوکها را در لیست

۲بعدی دیگری به نام dcts ذخیره میکنیم و آن را برمیگردانیم.

```
def C(u):
    if u == 0:
        return (math.sqrt(2)) / 2
    return 1

def dct_2D(a):
    return scipy.fftpack.dct( scipy.fftpack.dct( a, axis=0, norm='ortho' ), axis=1, norm='ortho' )
    #return cv2.dct(a)
    """F = [[0] * 8] * 8
    for u in range(8):
        for v in range(8):
            c = C(u) * C(v) / 4
            sum = 0
            for i in range(8):
                for j in range(8):
                    sum += math.cos((2 * i + 1) * math.pi * u / 16) * math.cos((2 * j + 1) * math.pi * v / 16) * f[i][j][index]
            F[u][v] = c * sum * 10 ** 13
    return F"""
```

در شکل بالا، پیاده‌سازی dct_2D را مشاهده میکنید. خطوط کامنت شده‌ی قرمز تابع دستی پیاده‌سازی شده با استفاده از فرمول اصلی تبدیل DCT بود که با بررسی نتایج آن، به این جمع‌بندی رسیدم که دقت خیلی بالایی دارد اما این دقت چندان برای ما اهمیت ندارد و علاوه بر آن محاسبات را بسیار کند میکند. (مثلا در خیلی از بلاکهایی که جواب در روش دستی پیاده سازی شده در مرتبه 14-10 بود، با کتابخانه مقدار صفر میگرفت)

در تابع بعدی که quantization و round انجام میشود، ابتدا جداول luminance و chrominance را آورده و دوباره با پیمایش روی هر بلاک، مقدار فعلی بلاک در سه مولفه Y, Cb, Cr را تقسیم بر خانه مرتبط در جدول مربوطه میکنم و round انجام میدهم. این مورد بلاک ۸*۸ جدیدی به ما میدهد که در لیست list_kl ذخیره میکنم. سپس در تابع زیگزاگ، بردار ۶۴ عضوی از هر بلاک ما را میدهد که ۱ عضو اول آن را در بلاکهای جدا (DC) حساب کرده و در یک لیست میریزم. با تابع dpcm به صورت تفاضلی برای هر سه مولفه Y, Cb, Cr ذخیره و حاصل در یک لیست جدا (new_) ریخته میشود.

نهایتا با تبدیل تک تک عضوها و اعداد به مقدار باینری و کد کردن سایز آنها با hauffman coding آنها را با pickle ذخیره میکنیم.