۱. الف

۶	۶	۴	۶
۶	۷	۶	۷
۶	۶	۵	۵
۴	۵	۴	۵

۶	۷	۷	۶
۷	۴	۴	۵
۶	۳	۲	۴
۶	۲	۳	۵

۰	۳	۲	۳
۰	۰	۰	۰
۱	۱	۰	۱
۱	۰	۱	۰

۳	۲	۵	۷
۲	۲	۴	۶
۰	۳	۵	۵
۲	۳	۴	۵

در مرحله‌ی اول به ۴ ناحیه *split* می‌کنیم.

۶	۴	۶	۶
۶	۷	۶	۷
۶	۶	۵	۵
۴	۵	۴	۵

۶	۶	۷	۷
۷	۵	۴	۴
۶	۴	۳	۲
۶	۵	۲	۳

۰	۳	۲	۳
۰	۰	۰	۰
۱	۱	۰	۱
۱	۰	۱	۰

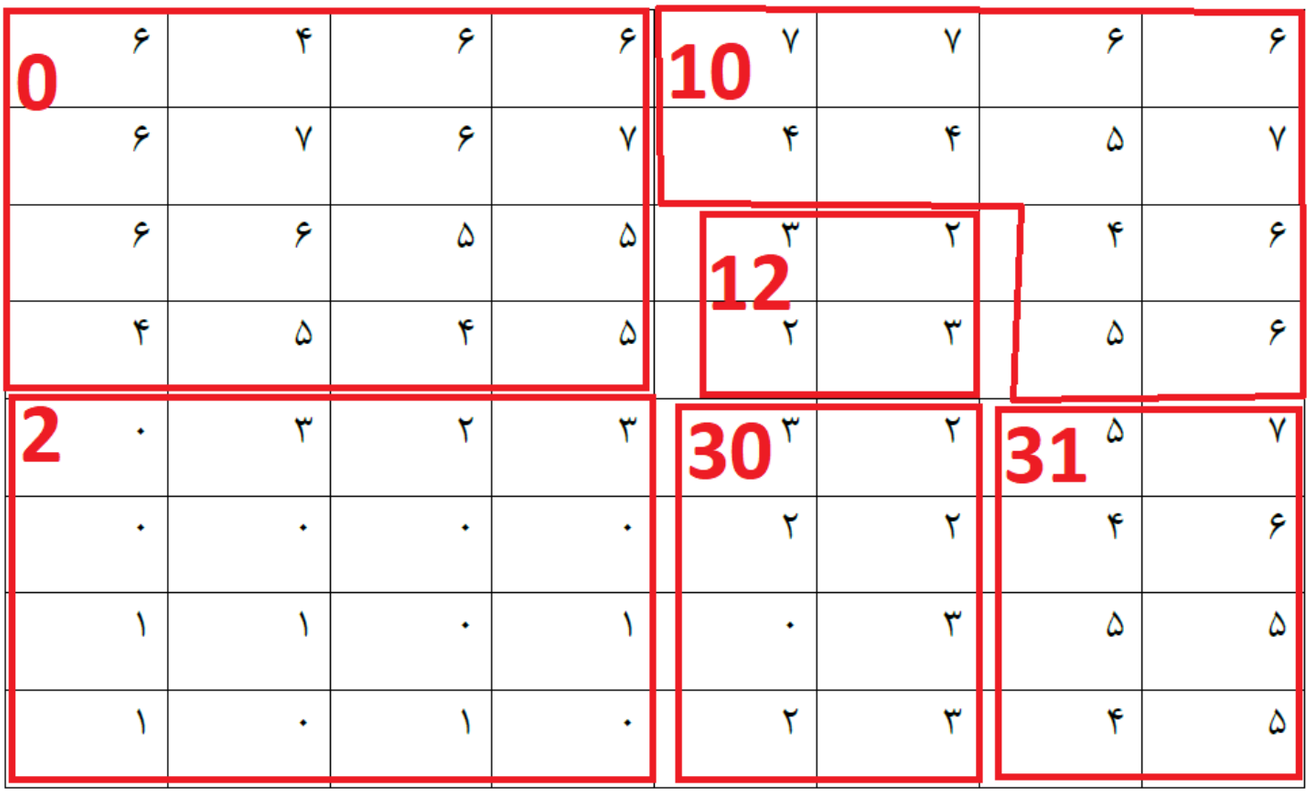
۳	۲	۵	۷
۲	۲	۴	۶
۰	۳	۵	۵
۲	۳	۴	۵

هر یک از ۴ ناحیه‌ی بوجود آمده را به ۴ ناحیه‌ی کوچکتر *Split* می‌کنیم.

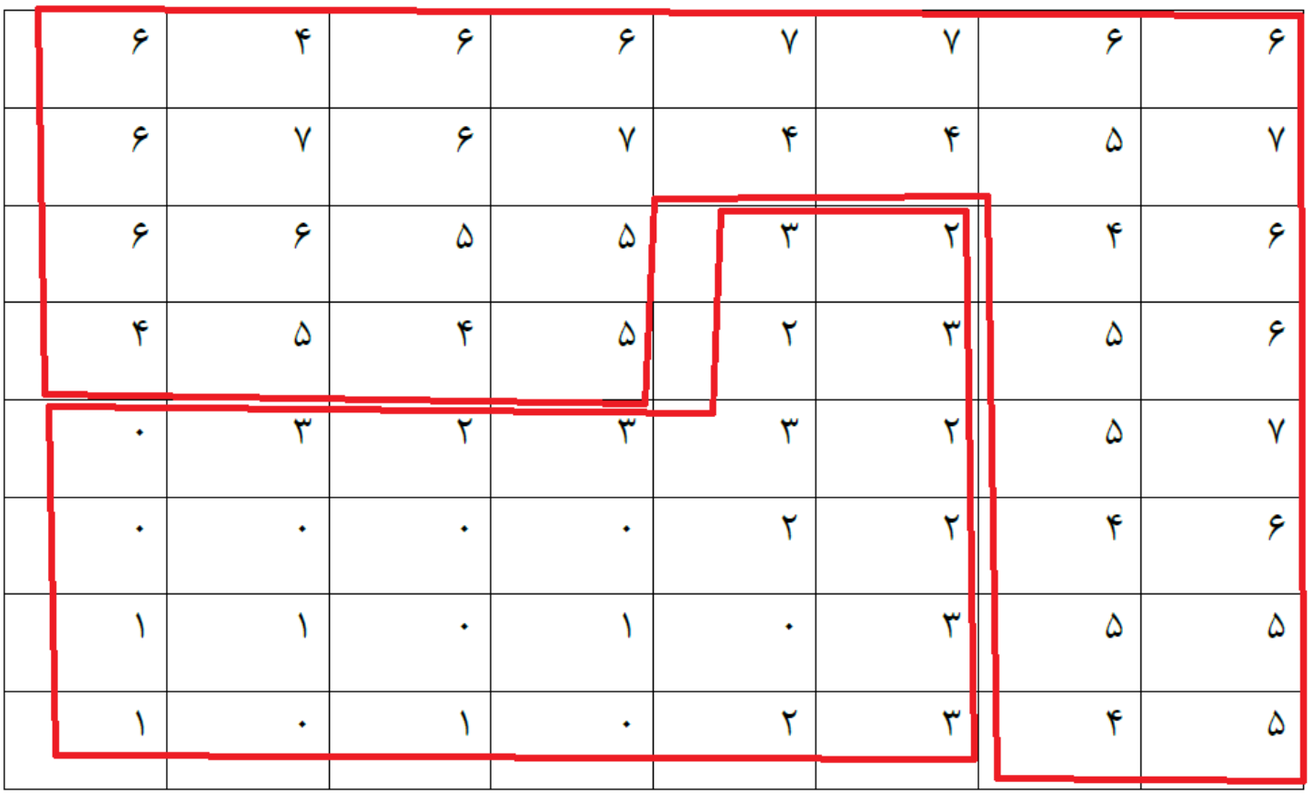
نواحی *00* و *01* و *02* و *03* قابل *merge* هستند. همچنین *20* و *21* و *22* و *23* هم قابلیت *merge* دارند. با *merge* کردن این نواحی، مرحله دوم الگوریتم به پایان می‌رسد چون هیچ ناحیه‌ای قابل *split*کردن نیست و هیچ دو ناحیه‌ای قابل *merge* کردن نیستند. پس به سراغ گام سوم می‌رویم و نواحی که از یک *parent* نیستند را *merge* می‌کنیم.

2. If any region R in the pyramid data structure is not homogeneous [$H(R) =$ FALSE], split it into four child-regions; if any four regions with the same parent can be merged into a single homogeneous region, merge them. If no region can be split or merged, go to step 3.

3. If any two adjacent regions R_i, R_j (even if they are in different pyramid levels or do not have the same parent) can be merged into a homogeneous region, merge them.



حالا نواحی *10* و *0* و *31* قابل *merge* شدن هستند و ناحیه‌های *12* و *30* و *2* هم با هم *merge* می‌شوند.



در نهایت کل شکل به دو ناحیه‌ی نشان داده شده در بالا تقسیم می‌شود که در هر ناحیه حداکثر اختلاف هر دو پیکسلی کمترمساوی ۳ است.

۱.ب

در روش *Region Growing* یک پیکسل به عنوان *seed* در نظر گرفته می‌شود و اگر پیکسل‌های مجاور با *seed* مشابه آن باشند به ناحیه‌ی *seed* اضافه می‌شوند. این کار تا زمانی بین پیکسل‌های ناحیه و پیکسل‌های مجاورشان تکرار می‌شود که دیگر هیچ شباهتی بین پیکسل‌های ناحیه و پیکسل‌های مجاور خارج از ناحیه وجود نداشته باشد.

در روش *Splitting and Merging* ابتدا کل ناحیه بصورت یک ناحیه در نظر گرفته می‌شود و سپس در صورتی که ناحیه همگن نباشد به ۴ ناحیه‌ی کوچکتر تقسیم می‌شود. (همگن بودن ناحیه را بصورتی می‌توانیم تعریف کنیم که هیچ دو پیکسلی در آن تفاوت محسوس نداشته باشند) در مرحله‌ی بعد دوباره همین کار را برای نواحی بوجودآمده کوچکتر تکرار می‌کنیم و اگر آن‌ها نیز همگن نباشند به ۴ ناحیه کوچکتر تقسیم می‌شوند و این کار تا آن‌جایی ادامه می‌یابد که هیچ ناحیه‌ی قابل *Split* کردنی وجود نداشته باشد (به بیان دیگر ناحیه‌ی غیرهمگن نداشته باشیم) از اینجا به بعد در الگوریتم *Merge* انجام می‌دهیم و ابتدا ناحیه‌هایی که والد یکسان دارند (یعنی از یک ناحیه *split* شده‌اند) و هیچ دو پیکسلی در آن دو ناحیه تفاوت محسوس ندارند را *merge* می‌کنیم و در نهایت وقتی هیچ ناحیه‌ی قابل *merge* به صورت گفته شده وجود نداشت سراغ نواحی می‌رویم که از یک والد یکسان نیستند ولی قابل *merge* شدن هستند یعنی هیچ دو پیکسلی از آن‌ها تفاوت محسوس ندارند (اختلاف هر دو پیکسلی از یک حد آستانه‌ای کوچکتر مساوی باشد)

تفاوت اصلی این دو روش در این است که *Region Growing* بصورت *bottom-up* ناحیه‌ی موردنظرش را می‌سازد یعنی از یک پیکسل بسیار کوچک شروع می‌کند و پیکسل‌های مشابه را به آن می‌چسباند تا به ناحیه‌ی بزرگتر برسد، در حالی که در *Split and Merging* نواحی بصورت *top-down* ساخته می‌شوند یعنی ابتدا کل تصویر را یک ناحیه می‌گیریم و آن قدر *Split* و *Merge* می‌کنیم تا به ناحیه‌های مشابه (که طبیعتا کوچکتر از کل تصویر هستند برسیم)

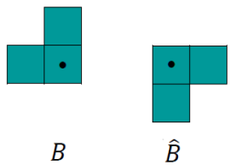
تفاوت دیگر این دو روش این است که در *Region Growing* فقط دنبال پیدا کردن یک ناحیه‌ی خاص هستیم که پیکسل‌های آن از شباهت با پیکسل *seed* بدست می‌آیند اما در *Split and Merging* می‌خواهیم کل تصویر را ناحیه‌بندی کنیم و برخلاف *Region Growing* در نهایت یک ناحیه بدست نمی‌آوریم بلکه تمامی ناحیه‌های تصویر خروجی الگوریتم است.

شباهت‌سنجی پیکسل‌ها در الگوریتم *Region Growing* بین یک نقطه و نقاط مجاور آن اتفاق می‌افتد (که می‌تواند بسته به نوع الگوریتم، نقطه *seed* و نقاط مجاور مرز ناحیه باشد یا اینکه خود نقاط مرز و نقاط مجاورشان) در حالی که شباهت‌سنجی روش *Split and Merging* بین تمام نقاط داخل یک ناحیه انجام می‌شود و بررسی می‌شود که حداکثر اختلاف دو پیکسل از ناحیه چقدر است.

۲. دوبار اجرای عملگر باز تفاوتی با یکبار اجرای آن ندارد (مگر اینکه عنصر ساختاری متفاوت در بار اول و دوم اجرا استفاده شود)

در عملگر باز وقتی ابتدا *erosion* انجام می‌شود تعدادی از نقاط حذف می‌شوند و برخی نقاط باقی می‌مانند وقتی *dilation* را اجرا می‌کنیم تعدادی نقطه اضافه می‌شود که به دلیل اینکه عنصر ساختاری بصورت انعکاس‌یافته در *dilation* استفاده می‌شود نقاط اضافه شده قطعا زیرمجموعه‌ای از نقاط تصویر اولیه هستند که در مرحله‌ی *erosion* از تصویر حذف شده بودند. وقتی برای بار دوم عملگر باز را اجرا کنیم هنگامی که *erosion* بزنیم همان نقاط *dilate*شده در مرحله‌ی آخر را حذف می‌کنیم، چون بقیه‌ی نقاط در مجاورت خود حداقل آن نقاط *dilate* شده را دارند و تحت *erosion* حذف نمی‌شوند. پس وقتی در نهایت *dilation* می‌زنیم به همین خروجی اجرای دفعه اول عملگر باز برمی‌گردیم چون *dilation* دوم دقیقا همان نقاطی را اضافه می‌کند که آخرین *erosion* حذف کرده است.

عنصر ساختاری و انعکاس‌یافته‌ی عنصر ساختاری:



۳.الف

- *Otsu* سریعتر از روش آستانه‌گذاری وفقی[۱] است چون فقط یک‌بار حد آستانه را بدست می‌آوریم ولی در آستانه‌گذاری وفقی لازم است برای تک‌تک پیکسل‌های تصویر این حد آستانه را بدست آوریم که به لحاظ پردازشی هزینه‌برتر است.
- برای تصاویر ساده با روشنایی مطلوب و کنترل شده روش *Otsu* پاسخ خوبی خواهد داد اما در حالت کلی و زمانی که قسمت‌های مختلف تصویر ممکن است تاریک و روشن باشند یا سایه افتاده باشند استفاده از یک حد آستانه برای همه‌ی نقاط تصویر نتیجه خوبی نخواهد داشت. روش آستانه‌گذاری وفقی با در نظر گرفتن همسایگی دور و بر پیکسل‌ها برای هر پیکسل یک حد آستانه بدست می‌آورد که مشکل گفته‌شده برای *Otsu* را حل می‌کند. پس می‌توان گفت در حالت کلی آستانه‌گذاری وفقی عملکرد بهتری دارد.

۳.ب

مراحل الگوریتم *adaptiveThreshold*

- ابتدا باید میانگین‌گیری انجام بدهیم که با توجه به اینکه *MEAN* می‌خواهیم یا *GAUSSIAN* کرنل مربوطه را تشکیل می‌دهیم و با کانولوشن گرفتن (*filter2d*) روی تصویر اعمال می‌کنیم تا یک ماتریس از مقادیر میانگین اطراف هر نقطه بدست بیاید.
- اگر پیکسل *i,j* تصویر را در نظر بگیریم، در خروجی با توجه به شرط *img[i][j]>mean[i][j]+c* در پیکسل خروجی ۰ یا ۱ می‌گذاریم.

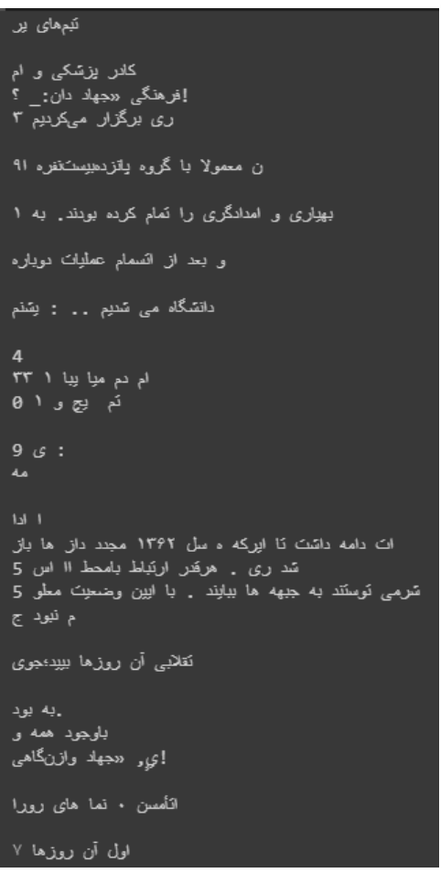
پارامترهای تابع cv.adaptiveThreshold (src, maxValue, adaptiveMethod, thresholdType, blockSize, C)

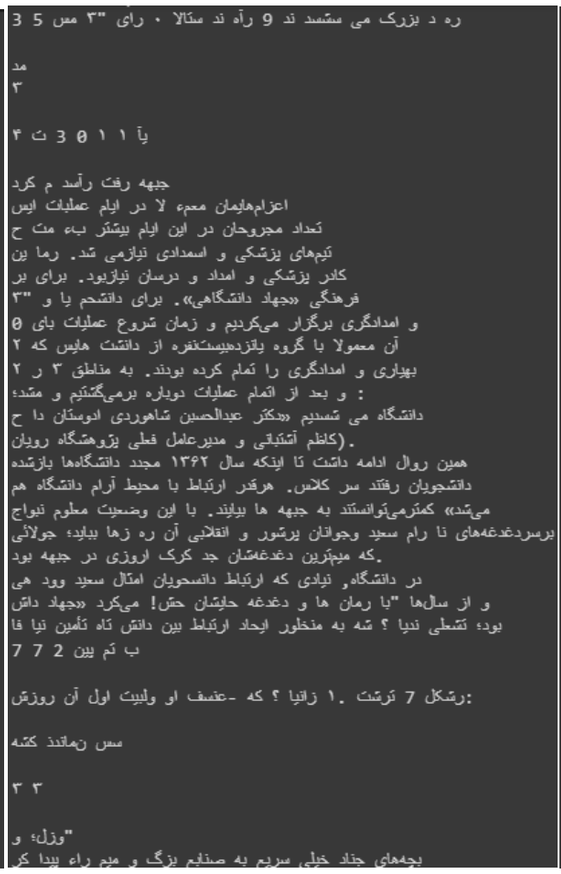
- *src* : تصویر ورودی
- *maxValue* : یعنی سطح دوم رنگ چند باشد و پس از اعمال آستانه‌گذاری چه مقداری برای سطح بالاتر در پیکسل‌های موردنظر گذاشته شود (سطح پایینتر که پیشفرض صفر در نظر گرفته می‌شود سطح دوم هم مثلا می‌تواند ۲۵۵ گذاشته شود)
- *adaptiveMethod* : روش *adaptive* که یا میانگین است (مقدار *MEAN*) و یا میانگین وزن‌دار گاوسی (مقدار *GAUSSIAN*)
- *thresholdType* : به معنی اینکه اگر از سطح آستانه کمتر بود صفر اختصاص داده شود یا سطح رنگ بالاتر، که اگر *THRESH_BINARY* گذاشته شود پیکسل کمتر از سطح آستانه صفر می‌شود و اگر *THRESH_BINARY_INV* گذاشته شود برعکس می‌شود یعنی اگر از سطح آستانه کمتر باشد سطح رنگ بالاتر را می‌گیرد و در غیراینصورت صفر خواهد شد.
- *blockSize* : مشخص می‌کند میانگین‌گیری در چه سایزی انجام بگیرد. به بیان دیگر ابعاد همسایگی موردنظر برای محاسبه حد آستانه را مشخص می‌کند.
- *C* : عدد ثابتی است که با حاصل میانگین‌گیری جمع می‌شود تا حد آستانه را مشخص کند.

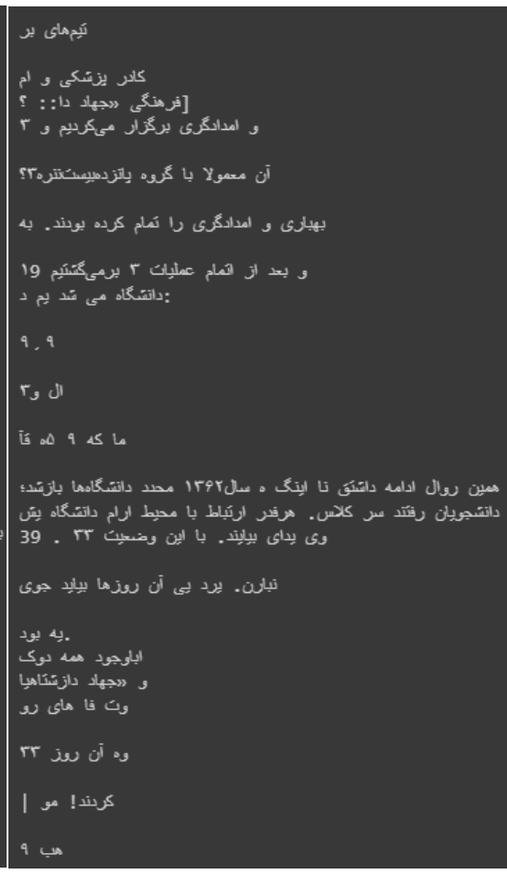
---

[۱] Adaptive Thresholding

۴.الف







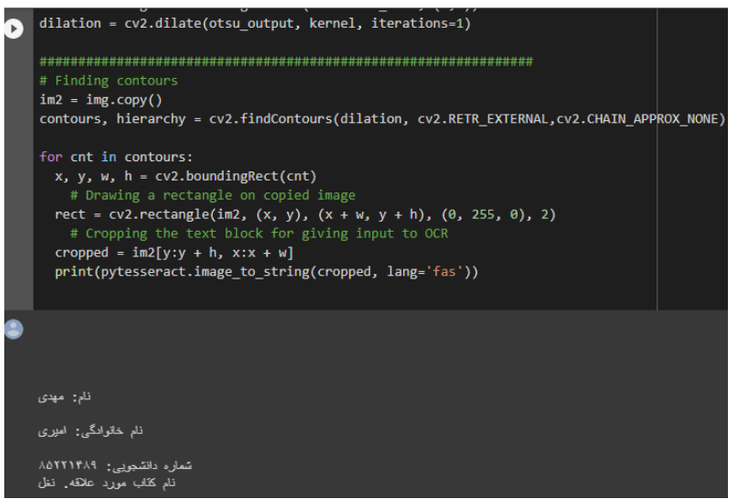
بدون استفاده از روش‌های *thresholding* با استفاده از پیش‌پردازش الگوریتم *adaptive* با استفاده از پیش‌پردازش الگوریتم *otsu*

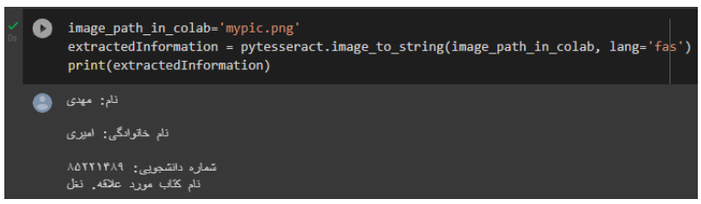
تصویر سمت راست بدون استفاده از روش‌های *thresholding* است و تصویر سمت چپ با استفاده از پیش‌پردازش *otsu*. همانطور که مشاهده می‌کنید بهترین نتیجه وقتی گرفته شد که از الگوریتم *adaptive* در پیش‌پردازش استفاده کردیم (تصویر وسطی) چون بیشترین جزئیات متن نسبت به دو حالت دیگر از تصویر استخراج شده است.

۴.ب

نام: مهدی
نام خانوادگی: امیری
شماره دانشجویی: ۹۸۵۲۲۱۴۸
نام کتاب مورد علاقه: نخل

تصویر اصلی





خروجی بدون پیش‌پردازش خروجی پس از اعمال پیش‌پردازش‌های آستانه‌گذاری و *dilation*

خروجی در دو حالت بدون پیش‌پردازش و با پیش‌پردازش تفاوتی نکرد که این می‌تواند به دلایل مختلفی مثلا نوع داده‌های متنی موجود در تصویر باشد.

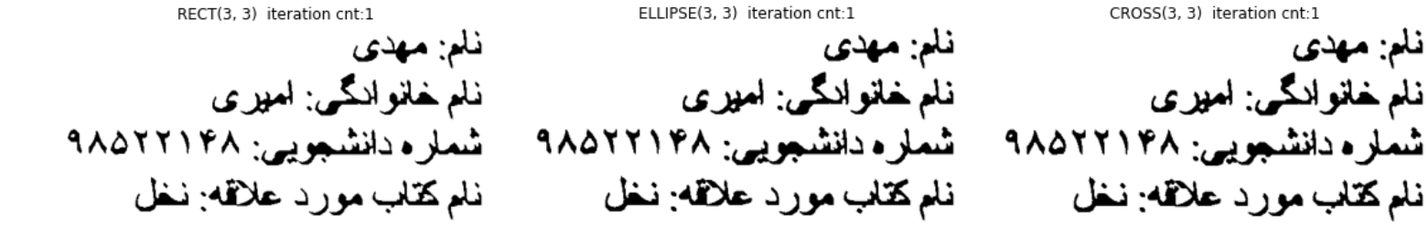
هدف از انجام پردازش مورفولوژی در اینجا حذف اضافاتی است که بدلیل اعمال افکت *paint stroke* به متن اصلی اضافه شده‌اند. چون پیکسل‌های متن ۰ هستند و پس‌زمینه ۱ است از *dilation* استفاده کردیم وگرنه استفاده از *dilation*(گسترش) برای حذف اضافات منطقی نیست. با *dilate*کردن پیکسل‌های ۱ گسترش می‌یابند و ۰های اضافی و کم‌اهمیت که دور و اطراف متن اصلی هستند از بین می‌روند.

تحلیل تاثیر عناصر ساختاری مختلف

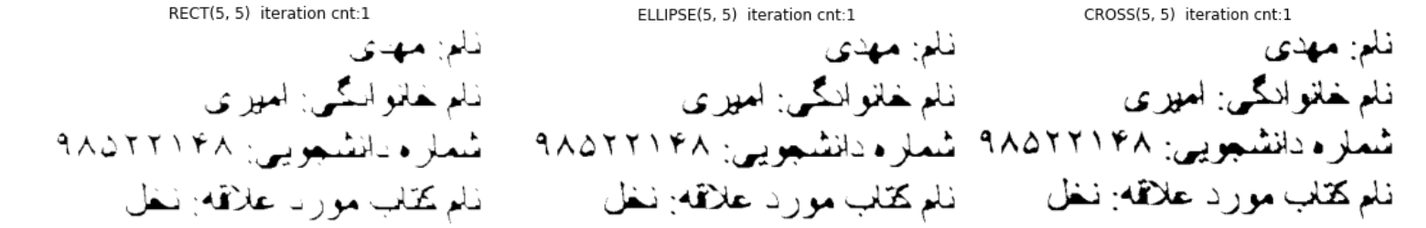
یک تابع کمکی پیاده‌سازی کردم که با عنصر ساختاری مورد نظر تصویر را گسترش دهد و بعد با *title* مناسب آن را نمایش دهد تا بتوانم حالت‌های مختلف را خروجی بگیرم و بررسی کنم.

سه شکل مختلف برای عناصر ساختاری در اختیارمان قرار دارد که مستطیل، دایره و صلیب هستند. از آن‌جایی که جزئیات اضافی افکت *paint stroke* نظم مشخصی ندارند سایز افقی و عمودی عنصر ساختاری را در همه‌ی حالت‌ها یکسان و متقارن گذاشتم.

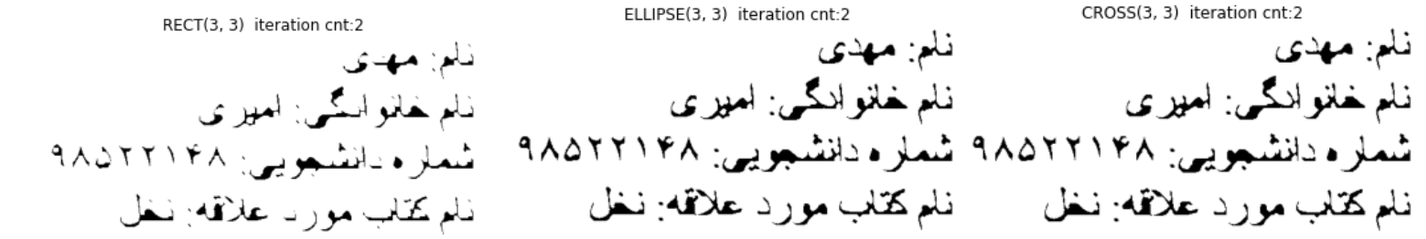
```python
def get_dilation_result(struct_shape, struct_size, iterations, shape_str):
    kernel = cv2.getStructuringElement(struct_shape, struct_size)
    dilation = cv2.dilate(otsu_output, kernel, iterations=iterations)
    plt.figure()
    plt.title(f"{shape_str}{str(struct_size)}  iteration cnt:{iterations}")
    plt.imshow(dilation, "gray")

get_dilation_result(cv2.MORPH_RECT, (3,3), 1, "RECT")
get_dilation_result(cv2.MORPH_ELLIPSE, (3,3), 1, "ELLIPSE")
get_dilation_result(cv2.MORPH_CROSS, (3,3), 1, "CROSS")

get_dilation_result(cv2.MORPH_RECT, (5,5), 1, "RECT")
get_dilation_result(cv2.MORPH_ELLIPSE, (5,5), 1, "ELLIPSE")
get_dilation_result(cv2.MORPH_CROSS, (5,5), 1, "CROSS")

get_dilation_result(cv2.MORPH_RECT, (3,3), 2, "RECT")
get_dilation_result(cv2.MORPH_ELLIPSE, (3,3), 2, "ELLIPSE")
get_dilation_result(cv2.MORPH_CROSS, (3,3), 2, "CROSS")

get_dilation_result(cv2.MORPH_RECT, (3,3), 3, "RECT")
get_dilation_result(cv2.MORPH_ELLIPSE, (3,3), 3, "ELLIPSE")
get_dilation_result(cv2.MORPH_CROSS, (3,3), 3, "CROSS")
```



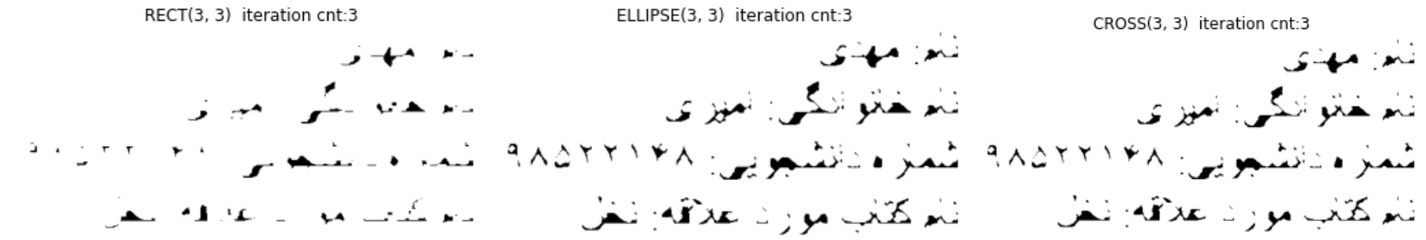
در حالتی که سایز ۳در۳ باشد و *dilation* فقط یکبار انجام شود، عنصر ساختاری مستطیلی بیشترین جزئیات اضافی را حذف می‌کند. با وجود اینکه تفاوت کمی وجود دارد ولی تا حدی حروف متن خروجی *RECT* نازکتر از دو حالت دیگر است.



در حالتی که عنصر ساختاری به سایز ۵در۵ برسد جزئیات اصلی متن هم در حال حذف شدن هستند بین سه شکل مختلف عنصر ساختاری با توجه به فضاهای خالی که در صلیب وجود دارد تا حدی کمتر از دو عنصر ساختاری دیگر جزئیات اصلی را از بین برده.



وقتی با همان عنصر ساختاری ۳در۳ دوبار عمل *dilation* را انجام دهیم باز هم جزئیات بیشتری را پاک می‌کنیم که در حالت مستطیلی این اتفاق به متن موجود در تصویر آسیب زده و در حالت صلیبی کمتر متن اصلی از بین رفته.



وقتی که سه‌بار یا بیشتر *dilation* را اجرا می‌کنیم تصویر به حدی لطمه می‌خورد که دیگر احتمالا نه‌تنها بهبود برای *OCR* حاصل نشده بلکه وضعیت بدتر هم شده و استخراج اطلاعات از آن سختتر هم شده.

منابع:

سوال ۱: Image Segmentation: Part 2. Deep dive into various image… | by Mrinal Tyagi | Towards Data Science

سوال ۳: Adaptive Thresholding with OpenCV (cv2.adaptiveThreshold) - PyImageSearch

سوال ۴: OpenCV: Image Thresholding

How to use image preprocessing to improve the accuracy of Tesseract (freecodecamp.org)

can't get pytesseract and cv2 to accurately OCR a screenshot from a phone (Python 3.8) - Stack Overflow

Erosion and Dilation of images using OpenCV in python - GeeksforGeeks