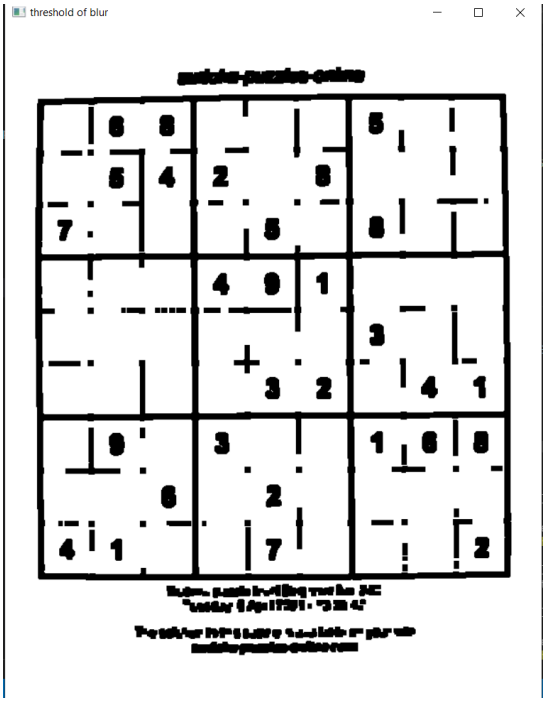
Bouruc Petru-Liviu
Grupa 334

Documentație – Tema 1

Extragerea informației vizuale din careuri Sudoku

Taskul 1 – Sudoku Clasic

Pentru început, odată cu deschiderea imaginii îi reduc din dimensiune prin intermediul funției *resize*, deoarece dimensiunea lor inițială este una foarte mare (3024x4032). Imediat după acest pas, fac o preprocesare pentru a determina colțurile careului sudoku, după care va urma să cropez imaginea. Astfel, la preprocesare transform imaginea în *grayscale* ca apoi să aplic un *medianBlur* apoi *GaussianBlur* pentru a obține o imagine sharpened combinata din cele 2. Acestea au rolul de a reduce noise-ul ca apoi sa putem aplica un threshold pentru a obține liniile mai pronunțate, pe care aplicăm *Canny* pentru edge detection. Inițial valorile kernel-ului pentru *medianBlur* și *GaussianBlur* erau de 5, dar acesta blura prea mult iar impreuna cu *threshold* si *erode* se putea întampla să apară discontinuități în conturul exterior, nemaiputând astfel fi identificat.




```python
def preprocess_image(image):
    image = cv.cvtColor(image, cv.COLOR_BGR2GRAY)
    image_m_blur = cv.medianBlur(image, 3)
    image_g_blur = cv.GaussianBlur(image_m_blur, (0, 0), 3)
    image_sharpened = cv.addWeighted(image_m_blur, 1.2, image_g_blur, -0.8, 0)
    _, thresh = cv.threshold(image_sharpened, 20, 255, cv.THRESH_BINARY)

    kernel = np.ones((5, 5), np.uint8)
    thresh = cv.erode(thresh, kernel)

    # show_image("median blurred", image_m_blur)
    # show_image("gaussian blurred", image_g_blur)
    # show_image("sharpened", image_sharpened)
    # show_image("threshold of blur", thresh)

    edges = cv.Canny(thresh, 150, 400)
    contours, _ = cv.findContours(edges, cv.RETR_EXTERNAL, cv.CHAIN_APPROX_SIMPLE)
    max_area = 0
```

Odată scoase edge-urile, putem aplica și *findContours* pentru a obține un array cu aceste contururi, prin care vom itera pentru a găsi cele 4 colțuri ale celui mai mare contur (se folosește metoda iterativă, ținând minte maxime locale până ajungem la cel global).

Având identificate aceste puncte, partea de preprocesare s-a terminat, urmând ca următorul pas să fie decuparea imaginii pentru a scoate grid-ul de sudoku efectiv din ea. Am ales să folosesc

metodele *getPerspectiveTransform* și *wrapPrespective.* Un avantaj al acestor metode este schimbarea de perspectivă pentru imaginile puțin rotite. Am încadrat noua imagine într-una de 500x500.

Pentru această noua imagine, creez linii orizontale și verticale la distanțe de 55 pixeli (sunt 9 casuțe la sudoku, astfel 500 / 9 = aproximativ 55 pixeli).

Pasul final este să iterez prin aceste linii și intersecții de liniii pentru a scoate *patch-urile* din imagine. Pentru fiecare patch, îl cropez cu 15 pixeli pentru a mă asigura că nu iau marginile și aplic un *threshold*, evidențiind astfel daca acesta conține sau nu un număr. Daca în imaginea obținută după threshold voi avea 0, înseamnă că acolo unde am avut numere la aplicarea funției pixelii corespunzători au fost transformați în pixeli negri, marcați cu 0. Așadar, știm că în căsuța respectivă există un numar și notăm corespunzător în soluție. Valoarea pentru threshold am obținut-o prin mai multe încercări, cea de 150 deservind cel mai bine toate cazurile de antrenare.

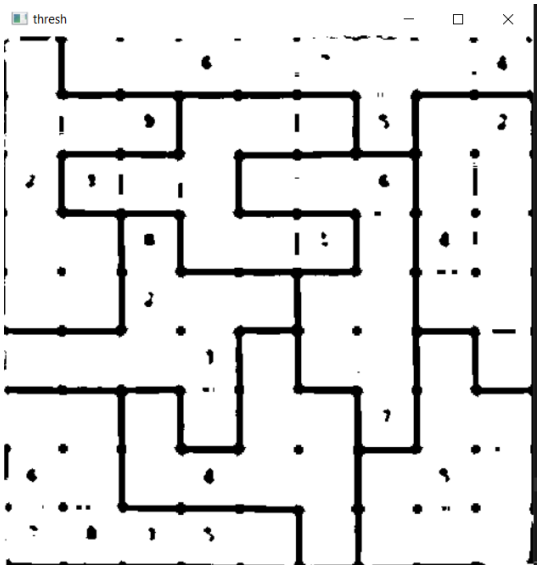
Menționez că pentru rezolvarea task-ului, am folosit codul din laboratorul 6.

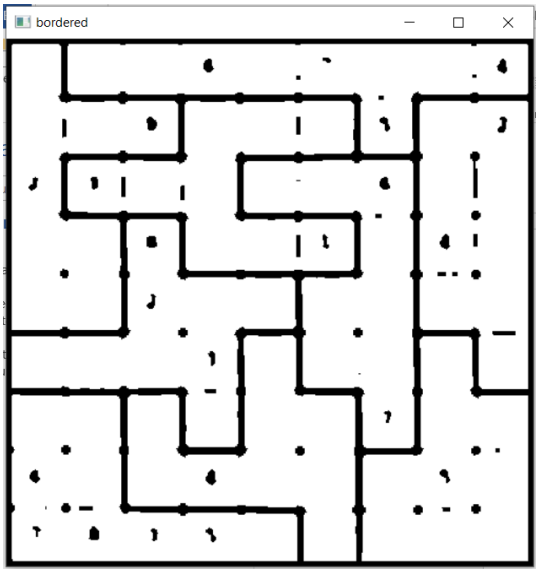
Taskul 2 – Jigsaw Sudoku

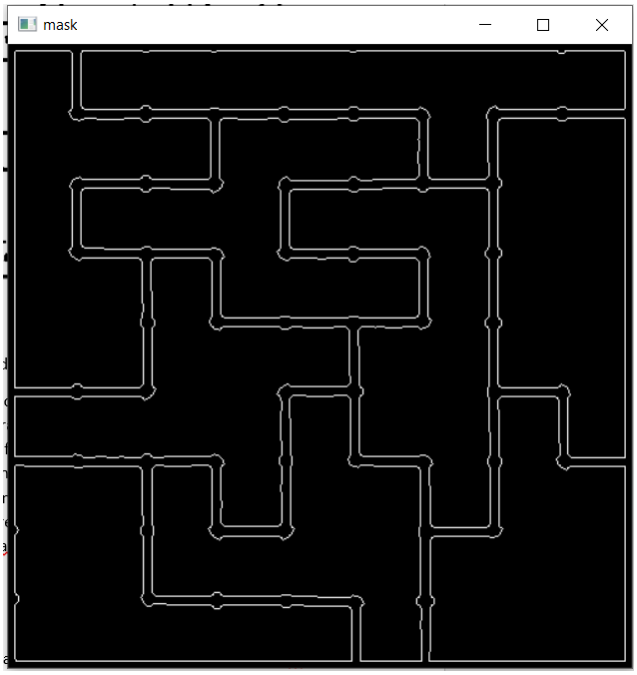
Asemenea primului task, am început acest task printr-un *resize* pentru a reduce dimensiunea imaginii, urmat de o preprocesare. Diferența la preprocesare este ca am lasat un blur mai mare aici, valoarea kernel-ului fiind 5, dat fiind faptul ca am linii mai ingroșate și mai subțiri, dar foarte important și faptul ca acum imaginile sunt și colorate. Asemenea primului task, am extras careul de sudoku pe care l-am încaadrat ulterior într-o imagine de 500x500.

De aici, scopul următorilor pași este să extrag contururile zonelor îngroșate. Astfel, am ales să transform imaginea în *grayscale* și să aplic un *medianBlur* de 7 pentru a fi sigur că scap de cât mai multe linii subțiri, rămânând astfel doar cu cele îngroșate. Mai mult, am aplicat un adaptiveThreshold de blocksize=31 si C=31. Am ales varianta adaptive deoarece, prin teste și datorită adaptivității la diferite zone, aceasta merge bine atât pe variantele color, cât și cele incolor. Valorile pentru blocksize si C au fost obținute prin test, cu mențiunea că odată ce creștem C-ul, valoarea pentru threshold se mărește, pierzând astfel din contururi. După aceasta, am aplicat operația de *opening* cu un kernel standard de (5,5), care reprezintă o dilatare urmată de erosion, cu scopul de a reduce din noise. De aici, am aplicat un border de 5 pixeli negru pe margini, pentru a putea extrage ulterior contururile.

Bouruc Petru-Liviu
Grupa 334



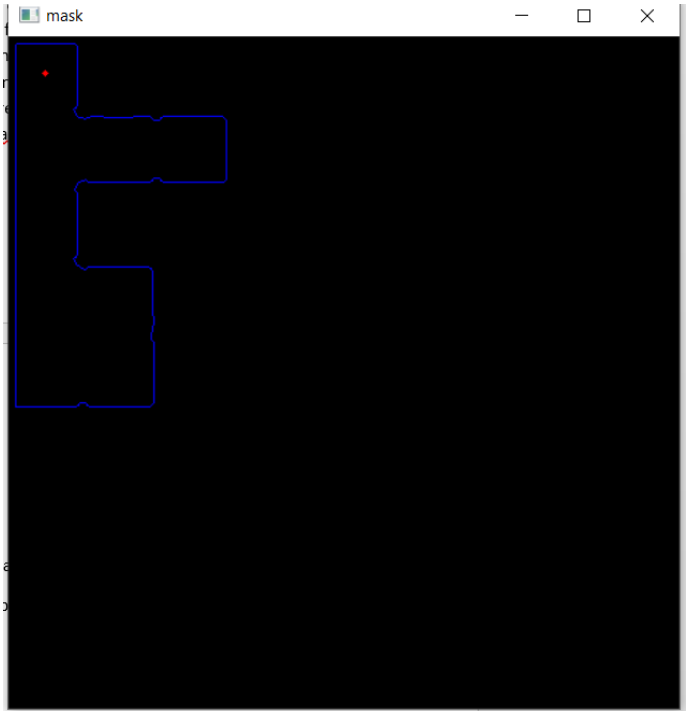


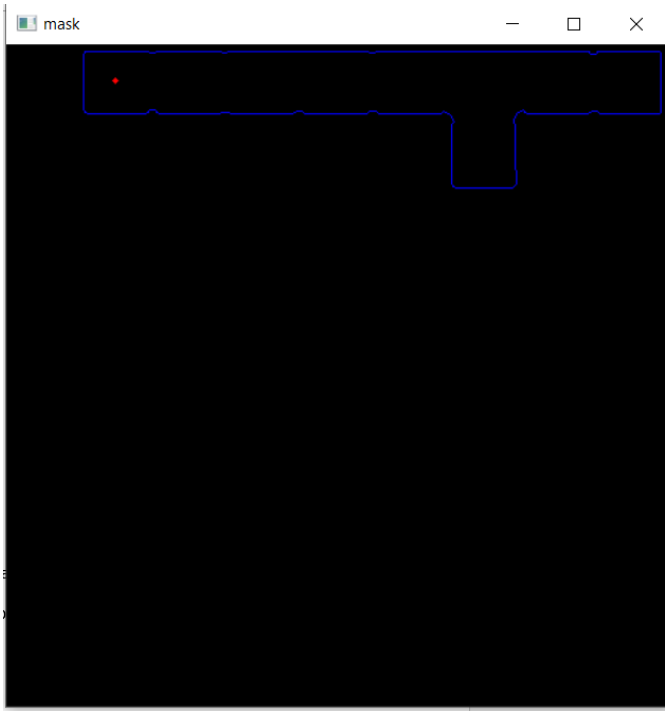


Pentru a extrage patch-urile, am folosit același procedeu ca la task-ul 1.

Parcurg patch-urile de la stânga la dreapta, de sus în jos. Pentru fiecare patch, am calculat punctul din mijlocul acestuia, și am iterat prin contururile extrase pentru a vedea în care se potrivește (am folosit funcția *pointPolygonTest* pentru a vedea daca se află în interiorul conturului sau în afara acestuia). Dacă se află în contur și acesta a mai fost “vizitat” în trecut, notez zona respectiva în soluție. Daca în array-ul *zones* zona respectivă are valoarea 0, înseamnă că aceasta nu a mai fost vizitată, fiind astfel prima dată când trecem prin ea. Îi asignez ultima valoare data unei zone +1. Numerotarea corectă a zonelor este respectată datorită modului de parcurgere a patch-urilor.

Bouruc Petru-Liviu
Grupa 334





Pentru determinarea existenței unui număr în patch, am aplicat același procedeu ca la task-ul 1.

Menționez că pentru rezolvarea task-ului, am folosit bucăți din codul din laboratorul 6.