Gépi látás beadandó

Kocsis Dávid – WS0LM1

Téma: Rendszám tábla felismerés: Rendszám tábla detektálása és leolvasása fényképről.

Tartalomjegyzék

[Bevezetés 1](#_Toc91092862)

[Technológia 2](#_Toc91092863)

[Algoritmus lépései 2](#_Toc91092864)

[Fejlesztői környezet meghatározása és használt külső könyvtárak 2](#_Toc91092865)

[Tesseract-OCR 3](#_Toc91092866)

[Nehézségek 4](#_Toc91092867)

[Algoritmus 4](#_Toc91092868)

[Tesztek 12](#_Toc91092869)

[Összegzés 14](#_Toc91092870)

[Források jegyzéke 16](#_Toc91092871)

# Bevezetés

Az automatikus rendszámtábla felismerő (ANPR) egy tömeges ellenőrzésre szolgáló módszer, amely optikai karakterfelismerést alkalmaz a képeken, így ismerve fel a járművek rendszámait. Felhasználása nem csak a közutakon, de még az egyetemünkön is elterjedt. Tipikus felhasználási célja a forgalom ellenőrzése autóúton, autópályán. Lehetőséget biztosít a szabálysértők (pl.: gyorshajtók) vagy körözött gépjárművek felkutatására. Egyetemünk belépési jogosultságokra használja, rendszámtábla leolvasása után, illetve az adatbázisban történő egyeztetés után nyílik fel a sorompó és hajthatunk be a megadott helyekre.

# Technológia

Az ANPR alapja a képmanipuláció. Első és talán legfontosabb lépése a rendszámtábla pozíciójának helyes detektálása. A rendszernek képesnek kell lennie egy képen meghatározni a rendszámtábla pontos helyét, ehhez egy állandót kell meghatároznunk, ami alapján tudjuk tervezni az algoritmust. Ez az állandó pedig nem más, mint a rendszámtábla alakja. Országoktól eltérően lehet más-más a rendszámtáblán lévő karaktersorozat, tehát erre nem hagyatkozhatunk. Különböző autómárkákon más-más helyen fordulhatnak elő a rendszámtáblák. Különböző célra használt járművek, vagy elektromos árammal hajtott autóknak pedig más színű a rendszámtáblája (pl.: taxi – sárga, 25 vagy 50 km-t megtételére képes tisztán elektromosan hajtott gépjárműveknek – zöld, diplomata gépjárművek - kék). Az egyedüli állandó, amit érdemes a képen az algoritmusnak keresni, az a rendszámtábla alakja, ami téglalap alakú poligon. Amennyiben a képen talál olyan poligont, ami lehetségesen egy rendszámtábla, akkor végrehajtja rajta a karakterfelismerést és megkapjuk a rajta lévő szöveget.

# Algoritmus lépései

1. Kezdeti kép beolvasása a fájlhoz vezető elérési út megadásával
2. Szürkeárnyalatos kép előállítása, majd zajcsökkentés és éldetektálás
3. Az éldetektálással kapott képen kontúr keresése és eltárolása
4. Lehetséges rendszám 4 sarokpontjának meghatározása
5. Maszkolás és rendszámtábla kivágása
6. Frontális kép meghatározása a kép átnyújtásával
7. Karakterfelismerés
8. Felismert karakterek visszaadása a felhasználó számára

# Fejlesztői környezet meghatározása és használt külső könyvtárak

A projekt fejlesztéséhez az órán megismert Thonny nevű programot használtam. Ez az IDE képek módosítására, illetve ablakokban történő folyamatos megjelenítésére kiválóan alkalmas. Programozási nyelvnek a már tanulmányaim alatt korábban megismert és alkalmazott Python nyelvet használtam a kifejezetten gépi látás projektek számára létrehozott külső könyvtárakkal.

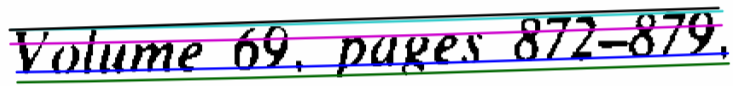
Használt könyvtárak:

* Opencv-python
* Numpy
* Imutils
* Pytesseract
* Matplotlib
* difflib

A könyvtárakból használt függvényeket a dokumentációmban később fogom kifejteni.

# Tesseract-OCR

Vonalkereséssel kezd, a vonal részének gondolt pixeleket eltárolja, majd vonalhoz rendeli, így megkapjuk a karakterek pozícióját a képen. Ezután vízszintes, „párhuzamos” vonalakkal körül határolja a szöveget. Ez nem teljesen igaz, csak a képen látható fekete vonal egyenes, a többi csak közelít hozzá, enyhén görbítettek. Itt a különböző magasságú karaktereknél húz vonalat, így tudja megkülönböztetni számokat, nagybetűket és különböző karaktereket.



*1. ábra – Tesseract-OCR szöveg körül határolása*

Ezután szétszedi karakterenként a már vízszintesen behatárolt szöveget. Következő lépés a lehetséges szavak megkeresése. Itt nagyobb „kihagyásokat”, vagyis szóközöket keres, így tudja behatárolni a szavak elejét és végét.

A képen vázlat látható

Automatikusan generált leírás

*2. ábra – Tesseract-OCR által behatárolt töréspontok*

Töréspontok mentén az egybeérő karaktereket szétszedi. Ezek prioritási sorrendben vannak egy rekurzív függvénnyel folyamatosan meghívva, ha egy vágás nem éri el megfelelő eredményt, akkor visszavonja majd megnézi a prioritás szerinti következőt. A hiányos karaktereket ezután megpróbálja következtetni. Mivel már megvannak a karakterek is, ezért összeveti az adatbázisban lévő adatokkal, majd visszaadja a karakterláncot, amit leolvasott a képről.[[1]](#footnote-1)

# Nehézségek

Számos nehézség lehetséges, amivel a szoftvernek meg kell tudnia birkózni. Ezek a következők:

* Gyenge képfelbontás nagy távolság miatt, vagy csak szimplán gyenge minőségű kamera miatt
* Elmosódott kép a mozgó autóról
* Kedvezőtlen fényviszonyok, árnyékok, tükröződések
* Több potenciális rendszám alakú poligon a képen
* Kicsi betekintési (rálátási) szög a rendszámra

# Algoritmus

Az algoritmus fő lépései a következő folyamatábrán jelennek meg:

*Folyamatábra az algoritmus fő lépésiről*



*3. ábra - Kiinduló kép*

A kód nagy részét egy nagy függvénybe ágyaztam be, ami paraméternek megkapja a képet és visszaadja a kinyert szöveget, ami remélhetőleg egy rendszámot tartalmaz. A kapott képen az első művelet, amit végrehajtottam, az a szürkeárnyalatosra alakítás volt. Ehhez a korábban importált opencv könyvtár cvtColor függvényét használtam, majd paraméterként megadtam a képet, illetve a COLOR\_BGR2RGB paramétert, ami így sikeresen elvégezte a műveletet. Ennek az eredménye látható a 4. ábrán.



*4. ábra - Szürkeárnyalatos előállított kép*

Ezután egy zajszűrést hajtottam végre a már szürkeárnyalatos képen, ehhez szintén az opencv könyvtárat használtam, azon belül is a bilateralFilter függvényt. Ennek paraméterként a képet kellett megadni, illetve három szám értéket, az én algoritmusomhoz a 11, 17, 17 kombináció passzolt a legjobban, de teszteltem még két-három másik kombinációt is. Ezeket különböző zajcsökkentéses projektekből vettem át. A bilateralFilter függvény minden egyes pixel intenzitását a környező pixelek súlyozott átlagából számolja. Ez a súlyérték Gauss eloszláson alapszik. A súly nem csak a pixelek Euklideszi távolságot veszi figyelembe, hanem a radiometriai különbségeket is. Ide tartozik a távolságok különbségei, intenzitásbeli különbség, színmélységbeli különbség stb..

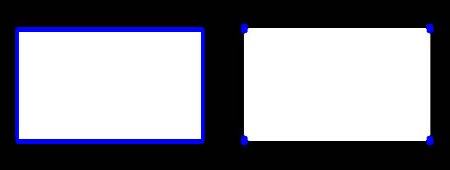
Az órán tanult Canny módszerrel éldetektálást hajtottam végre a képen, ez gyönyörűen kirajzolta a fő éleket a képen. A Canny módszer egy összetett algoritmus, lépései a következők: (Szürkeárnyalati konverzió), Gauss simítás, Differencia számítás, Nem-maximum vágás, Kettős küszöbölés, Hiszterézis küszöbölés.

Az órán tanult Canny módszerrel éldetektálást hajtottam végre a képen, ez hatékonyan kirajzolta a fő éleket, amelyek az 5. ábrán láthatók.



*5. ábra - Canny féle éldetektálás*

Itt jutottam el ahhoz a ponthoz, hogy már nem kell a képen semmiféle változtatást elvégeznem, csupán meg kell keresnem a rendszámtábla pozícióját. Kontúrkeresés következett az edged képen. Ehhez a findContours függvényt használtam, aminek meg kellett adni a képet, de mivel ez módosítaná, ezért először le kellett másolnom. Első paraméterének ezt adtam meg (edged.copy()). Második paraméter a kinyerési mód, ez nálam egy fa struktúra. A RETR\_TREE eltárolja az összes kontúrt és létrehoz egy család hierarchiát. Harmadik paraméter a körvonal közelítési metódus. A CHAIN\_APPROX\_SIMPLE, mint harmadik paraméter, arra szolgál, hogy memóriatakarékosan tárolja a kontúrokat. Egy négyzetnek például csak a négy sarokpontját tárolja el, mivel ez kellően elegendő információ a kontúr megrajzolásához. A kontúr pontjain végigmegy az algoritmus és az egy egyenesre eső pontoknak csak a két végpontját tartja meg.



*CHAIN\_APPROX\_SIMPLE működése*

A fenti képen látható, hogy a bal oldali téglalapon az összes pont eltárolásra kerül, míg a jobb oldali téglalapon csak a négy sarokpont. Mivel projektem során téglalap alakú poligonokat keresek, mint rendszámtábla, ezért ezt a tárolási módot választottam. Míg a bal oldali téglalapnak 734 pontját tárolja el, addig a jobb oldalinak mindössze a négy sarokpontját, vagyis négyet. A detektált kontúrokat ráhelyeztem a szemléltetés kedvéért a kiinduló képre, ez látható a 6. ábrán.



*6. ábra - Detektált kontúrok a kiinduló képre helyezve*

A körvonalakat ezek után rendeztem a sorted függvény segítségével, ennek először meg kell adni a kontúrokat, utána a rendezési szempontot a key funkcióval, majd a csökkenő sorrendet. Az eredményhalmazból az első 10-et választottam ki, ezekben keresett a továbbiakban a program.

Ezután egy ciklus segítségével a olyan kontúrt kerestem, amelynek pontosan 4 sarokpontja van. Ezeknek a sarokpontoknak a koordinátáit eltároltam egy erre létrehozott változóban. Másik eshetőség, ha nem találja meg ezeket a sarokpontokat, ezesetben nem találja meg a rendszámot sem. Ekkor egy egyszerű ellenőrzést hajt végre a program a koordinátákra, mégpedig, hogy hányat tárolt el. Ha ez nem négy, akkor itt le is áll, nem fut tovább feleslegesen, mivel nem lenne mit leolvasni.

Létrehoztam egy location változót, aminek kezdeti értékét none-ra állítottam. For ciklus segítségével végig megyek a kontúrokat tartalmazó tömbön, majd, ha megvan a 4 sarokpont akkor a break kulcsszóval leállítom. Ezt a négy sarokpontot eltárolom a location változóban. Az approxPolyDP függvényt használtam a cikluson belül, ami a Dougles-Peucker algoritmust használja, amelynek a célja, hogy a pontokat hagyjon el egy ívből, de mégis közelítőleg hasonló ívet kapjon kevesebb ponttal. Ez a függvény közelíti a poligont a kontúrból, minél nagyobbra állítjuk a második argumentum értékét, annál jobban fogja egyenesnek kezelni a „recés” részeket.

Mivel megvan a 4 sarokpont, nem volt más dolgom hátra, mint létrehozni egy képet, amin ez a 4 sarokpont által közbezárt terület ki van emelve valamilyen módon. Ehhez egy maszkot használtam, mivel megvannak a sarokpont koordinátái, ezért könnyedén létre tudtam hozni a rendszámtábla kiemelését. A numpy könyvtár segítségével létrehoztam egy nullákkal feltöltött maszkot. Létrehoztam egy new\_image változót, aminek a drawContours() függvény segítségével megadtam a nullákkal feltöltött maszkot és a korábban meghatározott location változóban eltárolt koordinátákat. A rendszámtábla kirajzolását a 7. ábra szemlélteti.



*7. ábra - Detektált poligon kiemelése a kiinduló képen*

A numpy bitwise\_and() függvényével a képre ráillesztette a maszkot, így a kiinduló képünkből most csak a rendszámtábla látszódik, de ez még nincs körbe vágva, ezért a kép megtartotta az eredeti nagyságát, de a kép nagy része így fekete. Pár soros kóddal orvosoltam ezt a problémát, csupán a maszk legmagasabb, legalacsonyabb és két irányban legszélesebb pontját kellett venni, így minimalizálni tudtam a felesleges részt, ez látható a 8. ábrán.



*8. ábra - Maszkolt rendszámtábla*

Ahhoz, hogy növeljem a hatékonyságát a rendszernek, a korábban kapott, kivágott, rendszámtáblát visszanyújtottam egy téglalappá, úgynevezett 4 pontos transzformációt hajtottam végre rajta, így a különböző éles szögből fotózott táblákon jobban érzékeli a karaktereket a pytesseract függvénykönyvtár leolvasó függvénye. Ehhez két függvényt írtam, az egyik az order\_points, másik a four\_point\_transform.

Először az order\_points függvényt kell meghívni (ez a four\_point\_transformon belül kerül meghívásra), majd a location változóban eltárolt sarokpontok koordinátáit átadni neki. Ehhez először egy pts változót hoztam létre, amely egy np.array típusú változó és float32 értékeket tárol. Ennek a függvénynek a lényege, hogy egy rect tömbbe visszaadja a megfelelő sorrendben a koordinátákat. A sorrend a következő: bal felső sarok, jobb felső sarok, jobb alsó sarok és végül a bal alsó sarok. Miután ezeket a pontokat már jó sorrendben visszaadta a four\_point\_transform függvénynek, a következő képlet segítségével kiszámolja az új téglalap szélességét és magasságát:

Leghosszabb szélesség meghatározása:

ahol és az egyik sarokpont x és y koordinátája, és a másik sarokpont koordinátája.

Magasság kiszámolásához is ezt a képletet használtam, csak más-más sarokpontra cseréltem az a és b sarokpontok koordinátáit. A leghosszabb magasság és szélesség meghatározása után megvan a „keret”, már csak az opencv segítségével beillesztettem a kivágott képet. A beillesztés után a 9. ábrán látható visszanyújtott képet kaptam.

**

*9. ábra - Szemből nézetes rendszámtábla*

Mivel most már a legélesebb, de még detektálható szögből készített képet is át tudtam alakítani frontális nézetre, így már nincs más hátra, mint leolvasni a rendszámtábláról a karaktereket. Ezt megelőzően érdemes volt még egyszer alkalmazni a zajcsökkentést egy magasabb értékkel, hogy még sikeresebben végezze a dolgát a program. A pytesseract külső könyvtárat használtam a leolvasáshoz, annak is az image\_to\_string függvényét. Ezt a –psm 11 configurációval használtam, aminek a lényege, hogy minden karaktert leolvas a kivágott képről. Vannak konfigurációk, amik egybefüggő karakterláncokat keresnek, de ezek nem adtak jó eredményt számomra. Miután leolvasta a karaktereket, eltároltam egy text változóban. Mivel minimalizálni akartam a leolvasás utáni hibákat, ezért a véletlenszerűen leolvasott karaktereket megpróbáltam törölni. Ezt a következőképp hajtottam végre. Meghívtam a strip() függvényt, ami törölte a szóközöket. Mivel minden rendszám csakis alfanumerikus karaktereket tartalmazhat, ezért egy ciklussal töröltem, amelyik karakter nem volt alfanumerikus. Ezzel el tudtam kerülni a külföldi rendszámokban lévő kör alakra hasonlító karakterek leolvasását. Erre a rendszámtáblára szolgál példaként a 10. ábra.

A képen szöveg, eszköz látható

Automatikusan generált leírás

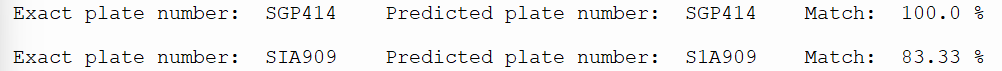
1. *ábra - Példa külföldi rendszámtáblára, amely félrevezetheti a leolvasást*

Előfordult olyan hiba is, mint például kisbetű lett leolvasva a nagy helyett. Mivel minden rendszámtábla csak nagybetűt tartalmazhat, ezért az upper() függvénnyel a kicsiket naggyá alakítottam. Ezután nem volt más hátra, mint visszaadni a leolvasott karakterláncot.

# Tesztek

A korábban említett műveletek jelentős része egy nagy függvényben voltak megírva, ami csupán a kinyert stringet adja vissza. Célszerű volt függvényben csinálni a teszt miatt, így ezután egy for ciklussal folyamatosan meg tudtam hívni különböző képekre, amelyekből ki tudja nyerni a rendszámokat és összehasonlítani az általam leolvasott ténylegesen helyes rendszámmal. A gyűjtött rendszámaim fájl neveit átírtam a tényleges rendszámra. Ezek lesznek az elvárt értékek.

Első tesztemhez egy test\_script\_different\_cars() függvényt írtam, ami az elején deklarált 3 listát, az egyik a predicted\_list, itt lesznek sorban a leolvasott rendszámok, második a known\_plates, ami a fájlok neveit tárolja és ezek egyben a helyes értékek is. Harmadik a statistic lista, ami pedig a két különböző lista elemeit veti össze és eltárolja százalékos egyezést. Itt lehetett volna más adattípust használni, például egy szótárt vagy tuple-t, ami szorosabb kapcsolatot biztosít a listák között, de a teszt szempontjából a listák is tökéletesen megfeleltek. Az automatizálás szempontjából egy txt fájlban tároltam el a képek elérési útját, így egy for ciklussal folyamatosan be tudtam olvasni soronként. A beolvasás során egynumber\_plate\_text változóban eltároltam a numberplate\_to\_text függvény által visszaadott leolvasott rendszámot, az actual\_plate változóban pedig a helyes értéket. Mielőtt viszont ezt eltároltam volna, ezen végre kellett hajtanom egy darabolást, mert ez az elérési utat és a kiterjesztést tárolta, nem pedig csakis a rendszámot string formában. Ehhez írtam egy függvényt, ami paraméterként megkapta az inputpath változó értékét (ez a korábban txt fájlból beolvasott i. iteráció sora). Itt a replace függvényt használtam (kicseréli a régi szöveget, ami a karakterláncban megtalálható, az új szövegre), mivel egy mappában van az összes kép és az összesnek ugyan az a kiterjesztése, ezért könnyedén megadtam paraméternek a régi szöveget, ami a „rendszam kepek\\\\” volt, új szövegnek pedig egy üres sztringet: „”. Ugyanígy a „.jpg”-re is végrehajtottam majd visszaadtam a nyers rendszámot. Ezt eltároltam az actual\_plate változóban, majd az append metódussal hozzáadtam a known\_plates listához, mint következő listaelem. A leolvasott rendszámot hozzáadtam a predicted\_list-hez, így biztosítva, hogy egy indexen legyenek az egymással kapcsolatban lévő értékek, csak különböző listákban. Utolsó függvény amit írtam, a similar() függvény egy külső könyvtárat használ, nem mást mint a difflib-et. Erre a két sztring összehasonlításához volt szükség. Meghívása után a similar() visszaadott egy 0 és 1 közti értéket, ezt felszoroztam 100-al majd kerekítettem 2 tizedesre, hogy szép százalékos értéket kapjak. Ezt az értéket hozzáadtam a statistic listához. Szemléltetés kedvéért minden egyes sorra kiírtam a kívánt rendszámtáblát, a leolvasott rendszámtáblát, majd a százalékos egyezést. A 10. ábrán látható a teszt futása közben megjelenő szöveg egy része. Ezzel a megjelenítési móddal egyértelműen és átláthatóan tudtam visszaadni a leolvasás eredményét.



*10. ábra – Output részlet*

A mainben csupán egy statistic változóra van meghívva a test\_script\_different\_cars függvény, ez kiírja statisztikám átlagát. A 11. ábrán tekinthető meg a teszt eredménye.



*11. ábra – Output részlet*

Ez a teszt különböző gépjárművek leolvasására szolgált, de írtam egy másik jellegű tesztet is, ami egy és ugyan azon jármű rendszámát olvassa le, csak különböző szögekből. Ehhez egy hasonló függvényt írtam, mint a korábbi, csak mivel az összes rendszám ugyan az, ezért a fájlok nevét nem a rendszám adja, hanem egy növekvő számsorozat, így nem volt szükségem még egy txt-re. Egy for ciklussal könnyedén tudtam generálni sztringeket, amelyek az elérési utat tárolják egy listába.

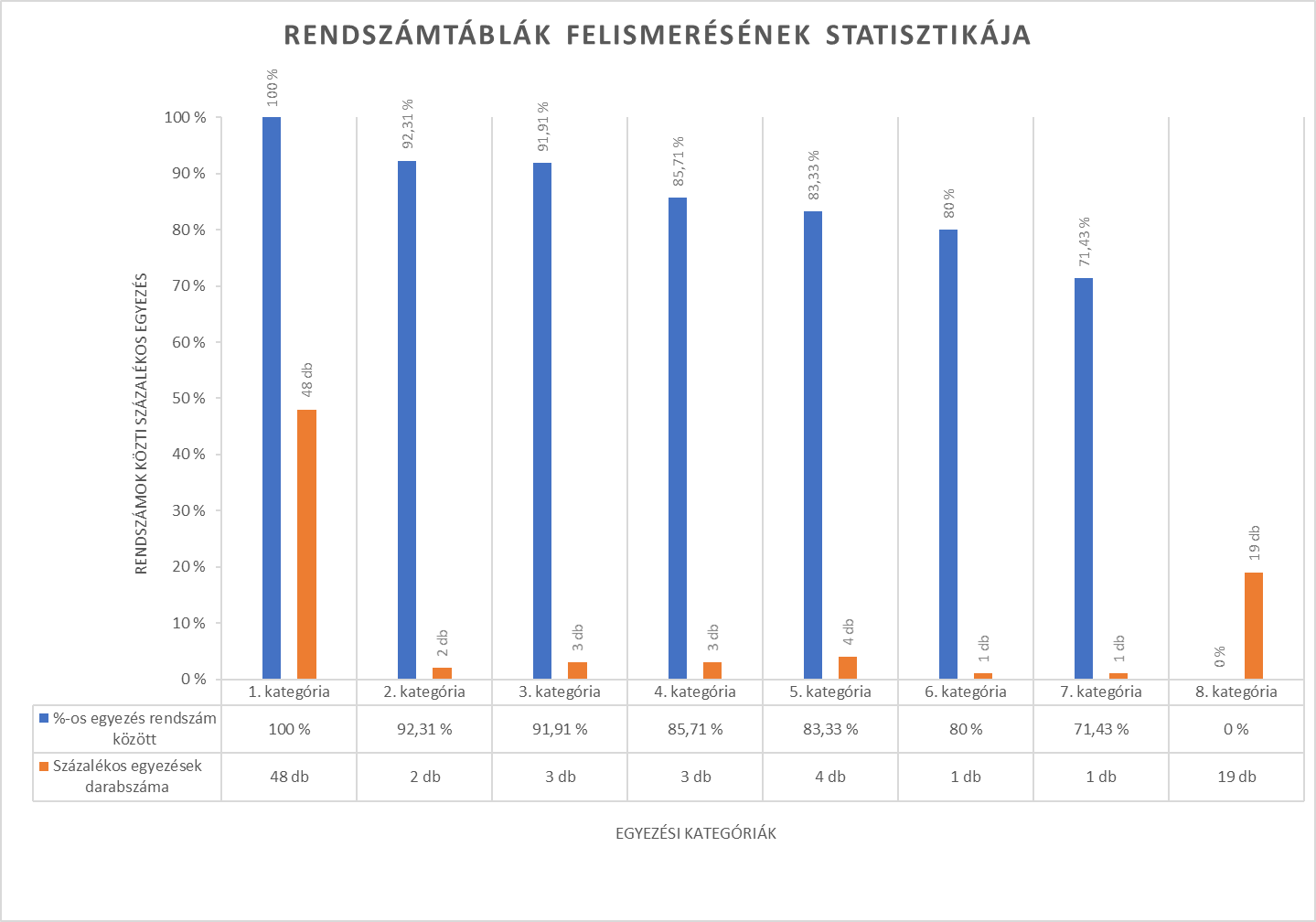
A képen szöveg, autó, kültéri, fű látható

Automatikusan generált leírás

*12. ábra – Rendszámtábla szembe forgatása*

A 12. ábrán látható igazán, hogy elengedhetetlen volt a rendszámtábla téglalappá való kinyújtása. Éles szögből is könnyedén jól látható rendszámtáblát sikerül létrehoznia a programnak. 16 különböző szögből 85%-os teljesítménnyel végzett a program.

# Összegzés



*12. ábra – Diagramm a rendszámtáblák felismerésének statisztikájáról*

A diagramon rendszámtáblák felismerésének statisztikája látható. Az X tengelyen 8 kategória van feltüntetve, mivel az egyezések ezeket a százalékokat adták, ezeket soroltam egy-egy kategóriába. Ezek az értékek a kék oszlopokkal vannak jelölve. Ezeknek a százalékos értékeknek az előfordulása, vagyis darabszáma megjelenítésére a narancssárga oszlopok szolgálnak. Az egyszerűbb leolvasás miatt egy adattáblát szúrtam be a diagramm alá, ebből egyszerűbben kivehetőek, hogy melyik érték hányszor fordult elő.

A diagramról leolvasható a projekt eredményessége. Kiderül belőle, hogy a 81 leolvasott rendszámtáblából 48-at tudott 100%-osan leolvasni és ténylegesen helyes eredményt visszaadni. Volt 19 eset, amikor 0%-os egyezést adott vissza, ez akkor fordul elő, hogyha nem tud semmit sem leolvasni a kivágott képről. Ebből arra lehet következtetni, hogy nem találta meg a képen a rendszámtáblát. Ha az egész adathalmazt nézzük, akkor 74,06% volt a hibátlanul leolvasott rendszámtábla. Érdekes vizsgálni azt az esetet, amikor nem vesszük figyelembe a 0 %-os értékeket. Ha ezeket mellőzzük és újra számoljuk az átlagot, akkor 96,76%-kal olvasta le hibátlanul a rendszámtáblát. Itt kiderül, hogy a programunk „gyenge” része a rendszám körvonalának megtalálása. Ez fejleszthető a későbbiekben úgy, hogy preseteket állítunk be, majd azokon a képeken, ahol nem talál rendszámtáblát, ott a különböző preseteket használunk és újrahívjuk a függvényt. Ez a preset lehet a zajcsökkentő függvény argumentuma, Canny detektor paramétere, vagy csak nem az első 10 kontúrt nézzük, hanem többet.

# Források jegyzéke

Hollósi János, órai jegyzetek

<https://drive.google.com/drive/folders/12MHB2pPib3Tz7S2_pm-yCKP3FPv-jD9h>

Ray Smith, An Overview of the Tesseract OCR Engine

[https://static.googleusercontent.com/media/research.google.com/hu//pubs/archive/33418.pdf](https://static.googleusercontent.com/media/research.google.com/hu/pubs/archive/33418.pdf)

OpenCV dokumentáció

<https://docs.opencv.org/3.4/d4/d73/tutorial_py_contours_begin.html>

<https://docs.opencv.org/4.x/d9/d8b/tutorial_py_contours_hierarchy.html>

<https://docs.opencv.org/4.x/d3/dc0/group__imgproc__shape.html#ga0012a5fdaea70b8a9970165d98722b4c>

<https://towardsdatascience.com/simplify-polylines-with-the-douglas-peucker-algorithm-ac8ed487a4a1>

1. Forrás: https://static.googleusercontent.com/media/research.google.com/hu//pubs/archive/33418.pdf [↑](#footnote-ref-1)