

USA sebességkorlátozó tábla felismerő

Strack Ákos*

*Electronic address: `akos.strack@gmail.com`; URL: `https://github.com/HunBug/SpeedLimitSignRecogniser`

Contents

I. Bevezetés	3
II. Módszerek	3
A. Módszer választása	3
B. Felismerési folyamat	4
1. Előfeldolgozás	4
2. Téglalap keresés	5
3. Sebességkorlátozó tábla észlelés	5
4. Korlátozás felismerés	6
5. Egyéb korlátozások felismerése	8
C. Eredmények	8
III. Implementáció	9
A. Fejlesztési környezet	9
B. Program szerkezet	10
1. SpeedLimitSignRecogniser	10
2. Recogniser	10
3. FileSource	11
4. SourceNormaliser	11
5. Segmentation	11
6. CandidateFinder	11
7. Detector	11
8. Recogniser	11
9. FeatureExtractor	12
10. Eredményekkel kapcsolatos osztályok	12
11. Segédosztályok	12
C. Program használata	12
IV. Összegzés	13
Hivatkozások	14



1. ábra. 25-ös korlátozó tábla felismerése, „School” kiegészítéssel

I. BEVEZETÉS

Az algoritmus feladata, hogy autókba szerelt kamerák képe alapján, kizárólag klasszikus képfeldolgozási eljárások használatával észlelje az egyesült államokbeli sebességhatároló táblákat és felismerje az azokon található korlátozás mértékét, esetleg egyéb kiegészítő korlátozásokat is. A fejlesztés során a képfeldolgozási eljárásokon kívül programozás-technikai szempontok is figyelembe lettek véve.

II. MÓDSZEREK

A. Módszer választása

A feladat kiírása alapján nem lehet mesterséges intelligencia módszereket használni, és bár a két terület határán lévő algoritmusokat lehetett volna használni, de érdekes kihívásnak találtam, hogy kizárólag klasszikus képfeldolgozó eljárásokat alkalmazzak. Ezek fényében a használható eszköztára jelentősen lecsökkent, és az elolvasott cikkek , és a rendelkezésre álló mintaképek alapján a kiindulási alapot a Real-time Recognition of U.S. Speed Signs [1] cikke képezte.

A választás azért esett erre a cikkre, mert a legtöbb M.I.-t nélkülöző algoritmus szín vagy forma alapú szegmentálást használ az előfeldolgozás során, és előzetes véleményem szerint a

szín alapú szegmentálást eleve nehezebb robusztussá tenni egy ennyire változatos környezetben (nappal, éjszaka, időjárási viszonyok, megvilágítások), másrészt a fehér táblára nehéz megbízható kritériumot találni, ami robusztusan elválasztja a háttértől. A forma alapú tábla keresők közül ez a cikk írt le olyan módszereket, amikkel az amerikai sebességkorlátozó táblákra jellemző tulajdonságokat hatékonyan fel lehet ismerni. Ez a cikk egy olyan módszert ismertet, ahol nem feltétlenül körcentrikus alakzatokat, esetünkben téglalapot lehet detektálni a gradiensek alapján. A cikkben módszerek bonyolultsága nem lépte át azt a szintet, amit érdemben ne lehetne implementálni az adott határidőn belül és az eredmények is elég meggyőzők voltak ahhoz, hogy egy kezdetleges implementációja is használható eredményeket adjon.

Mindezek mellett még szempont volt, hogy legyen olyan része a leírt algoritmusnak, amit nem valósítottak meg az OpenCV-ben, és érdemben lehet egy képfeldolgozó algoritmus „alacsony szintű” implementációját bemutatni, illetve néhány saját ötletet/módosítást is ki lehet próbálni.

B. Felismerési folyamat

1. Előfeldolgozás

Az előfeldolgozás rész feladata a bemeneti képek normalizálása és biztosítani, hogy a későbbi algoritmusok által várt előfeltételek teljesüljenek. A használt LISA [2] adatbázisban alapvetően két típusú forrás volt, a színes képek, aminek minőségét első körben használhatatlannak ítéltem ilyen feladatra, ezért csak az Audi rendszere által szolgáltatott nyers képekkel foglalkoztam. Mivel a használt algoritmusoknak nincs szüksége szín információkra, ezért itt a fő feladat az volt, hogy a Bayer mintás képből a lehető legjobb minőségű és felbontású szürkeárnyaltos képet állítsam elő. A demosaicing algoritmus a következőképp működik:

1. A forrás képet 4 felé bontani a 2x2-es Bayer minta pozíciói szerint.
2. Az így kapott (fele felbontású) képek intenzitásának egymáshoz igazítása. Ez histogram equalization-nel[3] történik meg.
3. A 4 rész-kép felskálázása eredeti méretre. Az interpolációhoz Lanczos[4] módszert

használok.

4. A 4 darab, eredeti méretű kép átlagolása.

Az adatbázisban található mintaképek alapján a Bayer minta különböző színeinek kiegyenlítésére a histogram equalization megfelelő eredményt adott. A 3. és 4. lépésben használt interpoláció és átlagolás hatására az eredmény kép már egyfajta zajszűrésen átmegy, de tapasztalatok alapján a későbbi algoritmusok jobb eredményt adnak, ha az előfeldolgozás végén egy Gauss simítást is használok még.

2. Téglalap keresés

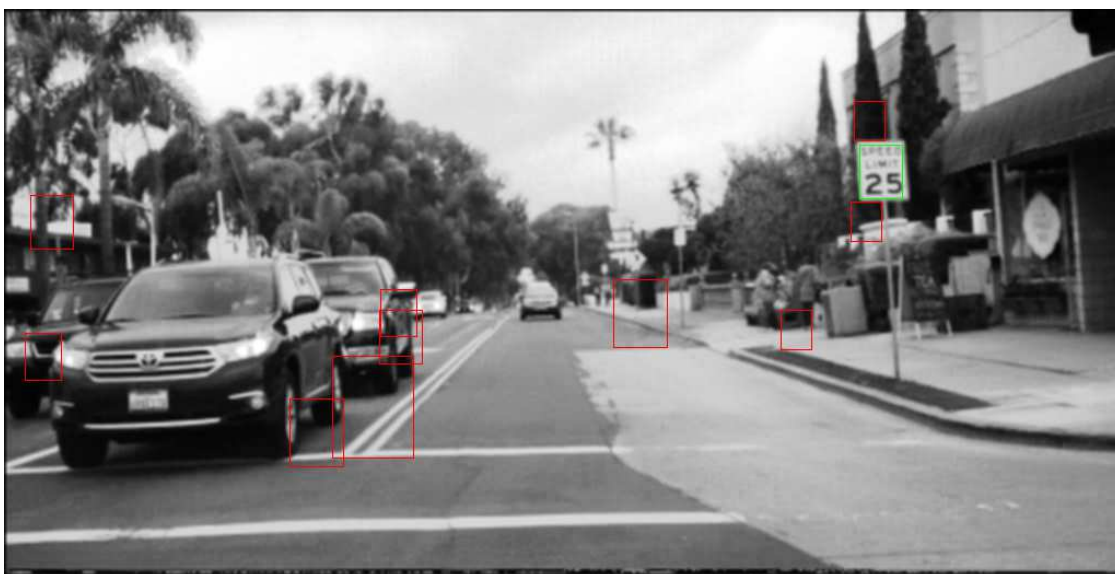
Ez a lépés egy előszűrésnek, egy szegmentációnak tekinthető, ahol a képet szétbontjuk téglalapra hasonlító, illetve nem hasonlító részekre. Ettől a lépéstől azt várjuk, hogy viszonylag gyorsan tudja azokat a helyeket a képen kiszűrni, ahol biztosan nem lehet tábla, így a későbbi, műveletigényesebb feladatoknak nagyságrendekkel kevesebb lehetséges helyet kell megvizsgálniuk, így rövidíthetjük a teljes feldolgozási időt, és a biztosan rossz helyek kiszűrésével a false positive detektálásokat is csökkenthetjük.

A téglalap keresési algoritmus lényege a kiindulási alapot adó cikkben [1] és annak hivatkozásaiban bőven ki van fejtve, itt csak az azoktól való lényegesebb eltérések emelném ki:

- Az algoritmus nem veszi pontosan figyelembe a a gradiensek irányát, csak függőleges és vízszintes irányokat ismer. Ezzel némi számítás spórolható meg, és tapasztalatok alapján ez is megfelelő eredményt ad
- A téglalap közepére leadott szavazatokat külön számolja mind a 4 irányba, felfelé, lefelé, jobbra, balra, és egy előre megadott küszöbértékkel külön-külön szűri a négy képet. Így azok a false positive találatok csökkenthetők, amik olyan képrészletekből adódnak, ahol a téglalapnak csak 1-2 oldala található meg.

3. Sebességkorlátozó tábla észlelés

Ebbe a fázisba kell megtalálni a sebességkorlátozó táblákat. Itt az alapul szolgáló cikkben [1] és a többi cikkben is általában valamilyen tanuló algoritmust használnak. Mivel az egyik



2. ábra. Lehetséges táblák jelölése. Pirossal amiket később elvetettünk, zölddel amiket valódi táblának detektáltunk.

célkitűzés az volt, hogy ilyenektől mentes legyen az algoritmus, ezért a cikk által javasolt Viola-Jones[5] alapú felismerő algoritmust sem használtam.

Első teszteket, ezen a téren talán jogosan legegyszerűbbnek nevezhető algoritmussal, a template matchinggel [6] végeztem. A normalizált, kereszt-korrelációs változatot használva értékelhető eredményt kaptam, és a fejlesztési idő rövideje miatt nagyon más algoritmust, vagy más feature-öket nem próbáltam ki. Tettem egy rövid kísérletet a SURF[10] és SIFT[9] algoritmusokkal is, de nem működtek jól, mert nem igazán találtak jellegzetes feature-t a mintaképen.

Az mintakeresőt csak azokon a helyeken futtatom, ahol az előző pontban leírt lépés lehetséges tábla helyet jelöl, így a futási sebesség jelentősen növelhető. Az adott potenciális tábla középpontnál, a várt maximális táblaméretnek megfelelő környezetben, különböző nagyságú mintákkal keresem a legjobb egyezést. Ha ez egy előre megadott küszöb felett van, akkor a jelezzük a tábla találatot, és a tábla pozíciójának pontos helyét.

4. Korlátozás felismerés

Az előzőekhez hasonló okok miatt itt is csak klasszikus képfeldolgozásban használt módszereket alkalmaztam. Mivel az észlelési részben is meglepően jó eredményt produkált a



3. ábra. Otsu binarizálása a táblának



4. ábra. Blobok keresése és szűrése

5. ábra. Számok keresése a táblán

legegyszerűbb algoritmus, ezért itt is megpróbálkoztam egy naiv módszerrel, a k-nearest neighbors-szal [7].

Ahhoz hogy a pixel alapú összehasonlítás működjön, ahhoz számokat pontosan meg kell találni a táblákon és a mintákat és a felismerni kívánt számokat egy méretre kell hozni, és normalizálni.

Az előző rész eredményeként egy viszonylag pontos helyét és méretét megkapjuk a táblának, így a számjegyek keresése viszonylag egyszerű, és a táblán belüli környezetre korlátozódik. Mivel ez a terület viszonylag tiszta képfeldolgozási szempontból, ezért egyszerű bináris képfeldolgozó módszerekkel jó eredmény érhető el. A számok megkeresésének lépései:

1. A tábla képének binarizálása. A küszöbérték meghatározására Otsu [8] módszere megfelelő a mi esetünkben.
2. Blob-ok megkeresése és azok tulajdonságainak kiszámítása.
3. A kiszámított tulajdonságok alapján biztosan rossz blobok kiszűrése. A szűrési feltételek a blob méretéből, oldalarányból és területéből adódnak össze.
4. Mivel az összes sebességhatárítás két számjegyből áll, ezért feltételezhetjük, hogy pontosan két blobot kell megtalálnunk. Ha kevesebbet találunk, akkor nem találtunk számot, ha többet találunk, akkor az előzetes szűrési feltételek nem voltak elegendőek.
5. Ha több mint 2 jelölt van még, akkor kiszűrjük a legkevésbé szám-szerűeket. Jelenleg azokat dobáljuk el, amik a táblán minél magasabban vannak. Amint csak 2 blob marad, akkor azokat jelöljük meg mint számok.

Ha megtaláltuk a számokat, akkor kNearest Neighbours algoritmussal megkeressük, hogy melyik előre elmentett szám mintára hasonlít legjobban, és az azon található szám értékét tekintjük a felismert sebességkorlátozásnak. Az összehasonlítás alapját az intenzitás normalizált pixelek adják.

5. Egyéb korlátozások felismerése

Az amerikai sebességkorlátozó táblákon elhelyeznek időnként egyéb korlátozásokat, vagy feltételeket is. Ezeknek felismerése valószínűleg hasonlóan történne, mint ahogyan az alap korlátozás felismerése, függetlenül attól, hogy milyen módszereket használunk az előző lépések implementációjához.

Ennek fényében, hogy megvalósítás szempontjából sok újdonságot nem mutat például az „School” felirat megkeresése, és az idő rövidsége miatt, csak jelzés értékkel került bele ennek megkeresése, ami a megtalált tábla felett, template matching [6]algoritmussal és egy megfelelő küszöbértékkel próbálja észlelni az iskola feliratot. A pozíciók számításánál érdemes lehet figyelembe venni a tábla dőlését is.

A kiegészítések felismerése közül a legnagyobb kihívást az jelentheti, mikor a tábla korlátozásai csak akkor érvényesek, ha a jelzőlámpa villog. Ehhez mindenképp szükséges több kép elemzése is, és a felvillanó lámpa felismerésére.

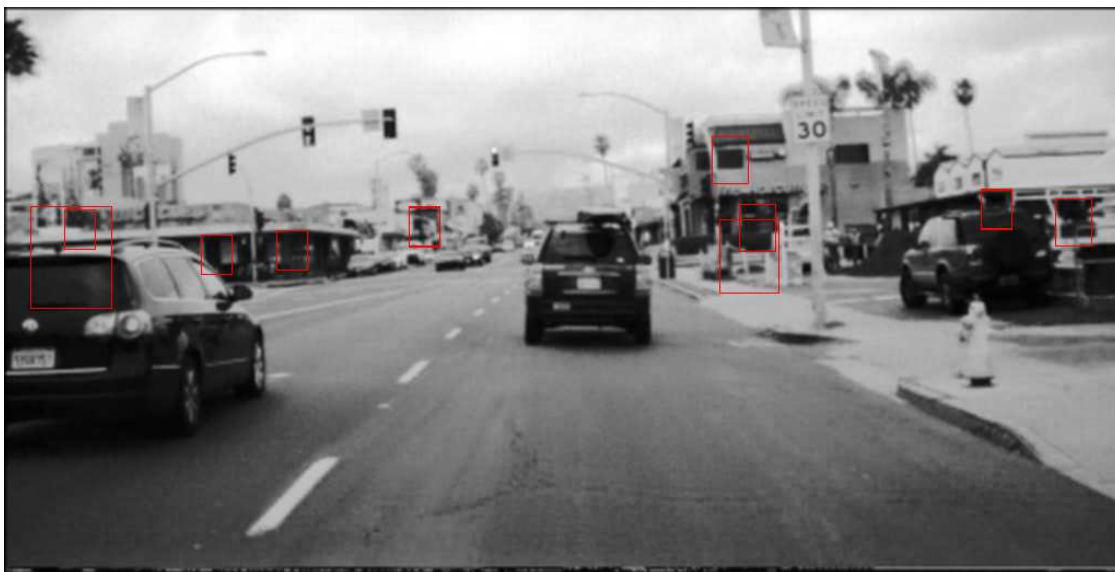
C. Eredmények

Az alábbi méréseket a LISA [2] adatbázis „aiua120306-1” könyvtárán végeztem. Az eredmények:

- 29 True Positive
- 1134 True Negative
- 8 False Positive
- 327 False Negative

Ami a következő teljesítmény mutatókat [11] adja:

- Recall: 0.0814607



6. ábra. Egy tipikus tábla tévesztés, mikor a tábla kerete beleolvad a környezetébe

- Precision: 0.783784
- F1 score: 0.147583

Az eredmények első ránézésre azt mutatják, hogy precision érték viszonylag jó, de a recall nagyon rossz. Ezek az eredmények azonban nem teljesen valósak, mert egyrészt a LISA annotációs fájlja nem teljes, például van olyan False Positive minta, ahol valójában tényleg van sebességkorlátozó tábla, csak az annotációs fájlba ezt nem jelezték, hanem egy másik, a képen szereplő táblát. A false negative szám még inkább távol esik a gyakorlati teljesítménytől, mert nincs figyelembe véve, hogy egy olyan kép-szekvencián, amin ugyan az a tábla van, ahogyan az autó halad az úton, akkor míg távol van a tábla, akkor az algoritmus nem ismeri fel, de ahogy közelebb ér a kocsához, akkor már igen. A távoli táblák jelenlegi kiértékeléskor a false negativ csoportba kerülnek.

III. IMPLEMENTÁCIÓ

A. Fejlesztési környezet

A teljes forráskód és felhasznált források és készült dokumentációk mind a git verziókövetővel vannak kezelve és a GitHub oldalon tárolva a <https://github.com/HunBug/SpeedLimitSignRecogniser> cím alatt.

A fejlesztés C++ nyelven történt, Eclipse fejlesztőkörnyezetben és MinGW fordítóval Windows 10-es és 7-es operációs rendszeren. A felhasznált programozási könyvtárak:

- Boost 1.59
- Opencv 3.0

Fejlesztés alatt több gond is akadt az Eclipse IDE-vel, elég gyakran futott exception-be, és jelentette a fejlesztőknek. Windows 10 alatt, debuggolás közben rendszeresen lefagyott a teljes operációs rendszer. A három hetes fejlesztés idő alatt a MinGW fordító 3 bug-jába is sikerült belefutni, amit különböző kerülő módszerekkel lehetett csak orvosolni.

B. Program szerkezet

A program egészére jellemző, hogy a különböző feldolgozási lépéseket, amiket az előző fejezetben külön szekciókban tárgyaltam, azok a program kódban külön-külön interfész osztályt kaptak. Ez elsőre talán túlzásnak tűnhet egy prototípus programnál, de eddigi tapasztalataim szerint igazából pont ebben a fejlesztési stádiumban igazán kifizetődő. Ellentétben azzal, mikor egy termék már kezd a végleges változathoz közeledni, akkor viszonylag keveset változik már a szerkezete a programnak, de a kutatási időszakban rengeteg kombinációt jó kipróbálni, és az automatikus (paraméter) optimalizáló és tesztelő rendszerek is könnyebben felépíthetők, hogy könnyen változtatható a processing flow, akár futási időben is.

1. SpeedLimitSignRecogniser

A main függvény helye, itt történik a program paraméterek beolvasása, majd a feldolgozás indítása.

2. Recogniser

Jelenleg ez fogja össze a feldolgozási folyamatot, és ez végzi el a eredmények kiértékelését is az annotációs fájl alapján.

3. *FileSource*

Egységes felület a fájlok, könyvtárak beolvasására.

4. *SourceNormaliser*

Az *ISourceNormaliser* interfészt megvalósító osztályoknak a feladata, hogy a különböző forrásból származó különböző tulajdonságokkal bíró képeket egységes formára hozzák, így a különbségekkel az algoritmusnak továbbiakban nem kell foglalkozni. a *DemosaicingNormaliser* osztály valósítja meg a „Előfeldolgozás”-ban leírtakat.

5. *Segmentation*

Szegmentáció, ami kizárhatja a biztosan „nem tábla”helyeket. Jelenleg nincs rendes megvalósítása, de például kivitelezhető lenne egy szín alapú előszűrés is.

6. *CandidateFinder*

Ezek az osztályok már egy pontosabb előszűrést végeznek. Ennek visszatérési értéke egy „vote-map”, ami a képen található pixelekhez tartozó találati szavazatokat tárolják. A „Téglalap keresés” szekció megvalósításai.

7. *Detector*

Az eddigi előfeldolgozások által megjelölt helyen végez pontos tábla keresést. Visszatér a talált táblák pontos helyével és méretével. A „Sebességkorlátozó tábla észlelés” szekció megvalósítása.

8. *Recogniser*

A *Detector* által megjelölt ROI-n belül keres számokat, és adja meg a korlátozás pontos mértékét. Jelenleg a kezdetleges „School” feliratkereső is itt van implementálva. A „Korlátozás felismerés” szekció megvalósítása.

9. *FeatureExtractor*

A osztályozó algoritmusokhoz használt feature készítő osztályok interfésze. Így egyszerűen változtatható mind betanítás és osztályozás közben a képekből kinyer feature-ök.

10. *Eredményekkel kapcsolatos osztályok*

RecognitionResult osztályba lehet tárolni a mért vagy elvárt mérési eredményeket. Az Evaluator osztály képes beolvasni az annotációs fájlok tartalmát, gyűjteni a mérési információkat, majd a mérés teljesítményt ezek alapján értékelni.

11. *Segédosztályok*

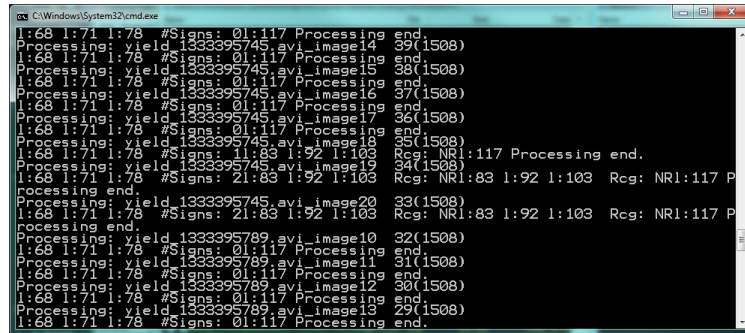
ImagingTools tartalmaz néhány, több helyen is használt egyszerű eszközt, ami a képfeldolgozási algoritmusok közben szükségesek lehetnek. DebugTools-ban a debuggoláshoz használt eszközök találhatók.

C. Program használata

A program egy parancssori alkalmazás. Mivel operációs rendszer függő dolgok nem lettek használva az implementáció során, ezért elvileg különböző operációs rendszerekre is lefordíthatónak kell lennie. Eddig csak Windows 7/10 rendszereken lett kipróbálva.

A parancssori argumentumok a következők:

- `-help`, megmutatja a program használatához a segítséget.
- `-f`, az utána megadott képfájl feldolgozása. Nem használható együtt a `-d` kapcsolóval
- `-d`, az utána megadott könyvtárban található összes „.png” képfájltra lefuttatja a feldolgozást. Nem használható együtt a `-f` kapcsolóval
- `-a`, az annotációs fájlt lehet megadni kiértékeléshez
- `-force`, kikényszeríti, hogy minden esetben számolja újra az eredményeket minden képhez. Ha ez nincs megadva, akkor megpróbálja betölteni a már elmentett eredményeket. Megszakadt könyvtár-feldolgozásnál vagy kiértékelési feladatoknál lehet hasznos.



7. ábra. Program futás közben

- -debug, debug képek mentése
- -separate, az eredmény képeket és adatokat szétválogatja a TruePositive, TrueNegative, FalseNegative, FalsePositive könyvtárakba

Egy tipikus használata:

```
SpeedLimitSignRecogniser.exe -d testFolder -a  
testFolder\frameAnnotations.csv -debug -separate
```

IV. ÖSSZEGZÉS

Viszonylag kevés fejlesztési idő alatt sikerült egy olyan demó programot készíteni, amiben demonstrálni lehetett alapvető programozási technikáktól kezdve a magas szintű képfeldolgozási könyvtárak használatán át egészen az alacsony szintű képfeldolgozó algoritmusok implementálásáig. Mivel ezeket a célokat nagyjából egyenlő súllyal vettem figyelembe, így egyik területen sem kapott kizárólagos figyelmet, így mindegyik részén lehetett volna jobb eredményeket elérni, ha csak arra részre lett volna minden energia fordítva. Ennek ellenére a program értékelhető felismerési eredményeket produkál, és továbbfejlesztése is könnyen megtehető, hisz kezdetektől fogva figyelembe lett véve ennek lehetősége.

Következő lépésben a egyes kódrészletek refaktorálása és tisztítása történik meg, hogy a további fejlesztések már teljesen letisztult kódbázisra épüljön.

Képfeldolgozás terén, a későbbiekben is használhatónak vélt részek, mint például a téglalap kereső algoritmus, optimalizálni kell mind minőség, mind gyorsaság szempontjából. Az eddig nem használt módszerek kipróbálása, mint például a cikkben [1]javasolt Viola Jones

[5] a felismerés pontossága és a sebesség javulása érdekében. Egyéb ötletek is felmerültek fejlesztés közben, mint például Super resolution, color segmentation, sign tracking az előfeldolgozás vagy egy validációs lépés során. Egyes esetekben a téglalap kereső nem működik, ha a táblakeret nem látható elég jól, és nem produkál megfelelő nagyságú gradienseket, ezekre az esetekre is kell megfelelő megoldást találni, például a lehetséges helyek keresésénél is a „Speed limit” szöveg keresése.

A kiegészítő korlátozások felismerését is implementálni kell.

Ezen kívül néhány segédeszköz fejlesztése is szükséges a hatékonyabb teszteléshez és kiértékeléshez. Ehhez az annotációs fájlokat is pontosítani kell, és megoldani, hogy egy videó szekvenciából származó képeket „egy táblaként” kezelje a kiértékelés. Teszteket egyéb adatbázisokon is érdemes lenne elvégezni.

Az algoritmust érdemes lenne átírni úgy, hogy az kihasználhassa a több szálú feldolgozást, illetve a GPU számítási teljesítményét. Mindkettő bizonyos szintig könnyen megtehető most is, hiszen jelenleg a különböző fázisok jól szeparáltak, így a pipeline egyes fázisai az egymás követő képeken egyszerre elvégezhetőek, illetve OpenCv maga is támogatja az OpenCL számításokat, ennek kihasználása nagyon kevés kódmódosítással megtehető. A saját, alacsony szinten megírt képfeldolgozó algoritmusokat is érdemes átültetni OpenCL-re, ha ezzel jelentős teljesítményjavulás érhető el a teljes feldolgozási folyamatra nézve.

-
- [1] Christoph Gustav Keller , Christoph Sprunk , Claus Bahlmann , Jan Giebel³ and Gregory Barattoff, „Real-time Recognition of U.S. Speed Signs”, Intelligent Vehicles Symposium, 2008
 - [2] LISA Traffic Sign Dataset , Laboratory for intelligent & safe automobiles, <http://cvrr.ucsd.edu/LISA/lisa-traffic-sign-dataset.html>
 - [3] Wikipedia, „Histogram equalization”, https://en.wikipedia.org/wiki/Histogram_equalization
 - [4] Wikipedia, „Lanczos resampling”, https://en.wikipedia.org/wiki/Lanczos_resampling
 - [5] P. Viola and M. Jones, „Robust real-time object detection”, Technical report, Compaq Cambridge Research Laboratory, Feb. 2001. CRL 2001/1.
 - [6] R. Brunelli, „Template Matching Techniques in Computer Vision: Theory and Practice”, Wiley, ISBN 978-0-470-51706-2, 2009
 - [7] Wikipedia, „k-nearest neighbors algorithm”, <https://en.wikipedia.org/wiki/K->

nearest_neighbors_algorithm

- [8] Nobuyuki Otsu, "A threshold selection method from gray-level histograms". IEEE Trans. Sys., Man., Cyber. 9 (1): 62–66. doi:10.1109/TSMC.1979.4310076., 1979
- [9] Lowe D G, „Distinctive image features from scale-invariant keypoints”. International journal of computer vision, 60(2): 91-110, 2004
- [10] Bay H, Ess A, Tuytelaars T, et al., „Speeded-up robust features (SURF)”. Computer vision and image understanding, 110(3): 346-359, 2008
- [11] Wikipedia, F1 score, https://en.wikipedia.org/wiki/F1_score