Detekcija registarskih tablica i raspoznavanje znamenki korištenjem OpenCV funkcija

Broznić, Adrian

Sveučilište u Rijeci – Odjel za informatiku

Preddiplomski jednopredmetni studij informatike

Adrian Broznić

Detekcija registarskih tablica i raspoznavanje znamenki korištenjem OpenCV funkcija

Završni rad

Mentorica : izv. prof. dr. sc. Marina Ivašić-Kos

Rijeka, 24.9.2021.

ODJEL ZA INFORMATIKU
SVEUČILIŠTE U RIJECI

Rijeka, 17.02.2021.

Zadatak za završni rad

Pristupnik: **Adrian Broznić**

Naziv završnog rada: **Detekcija registarskih tablica i raspoznavanje znamenki korištenjem OpenCV funkcija**

Naziv završnog rada na eng. jeziku: **License plate detection and digit recognition using OpenCV functions**

Sadržaj zadatka: Proučiti metode računalnog vida i obrade slika koje se koriste kod detekcije registarskih tablica i kod raspoznavanja znakova na tablicama.

Pronaći, implementirati i testirati različite metode za detekciju registarskih tablica i raspoznavanje znakova korištenjem C++ programskog jezika i biblioteke OpenCV-a. Objasniti algoritme iz biblioteke OpenCV korištene za detekciju registarskih tablica te ostale resurse korištene u razvoju aplikacije. Prikazati rezultate i prokomentirati ih.

Mentor

Izv. Prof. dr. sc. Marina Ivašić-Kos

Voditelj za završne radove

Dr. sc. Miran Pobar

Zadatak preuzet: 17.02.2021.

Sadržaj

Sažetak

U ovom radu ćemo opisati i usporediti uspješnost četiri različita algoritma za detekciju registarskih tablica te primijeniti algoritam za prepoznavanje znakova tablice na najuspješniji od ta četiri. Od četiri korištena algoritma jedan detektira moguće registarske tablice korištenjem Haar kaskada dok preostala tri pronalaze konture na slikama filtriranim pomoću OpenCV funkcija. Za klasifikaciju znakova koristiti ćemo neparametarsku metodu klasifikacije zvanu KNN (k-nearest neighbors) algoritam.

Ključne riječi : OpenCV, algoritam, KNN, Haar kaskade, konture, registarske tablice

Uvod

Detekcija i prepoznavanje registarskih tablica je zadatak iz područja računalnog vida koji ima široku primjenu u cestovnoj sigurnosti, bilo to radi praćenja auta koji ugrožavaju sigurnost u prometu ili olakšavanje ulaska u parking ili područja s rampom, ubrzavanja prolaska na autocestama i cestama s naplatom.

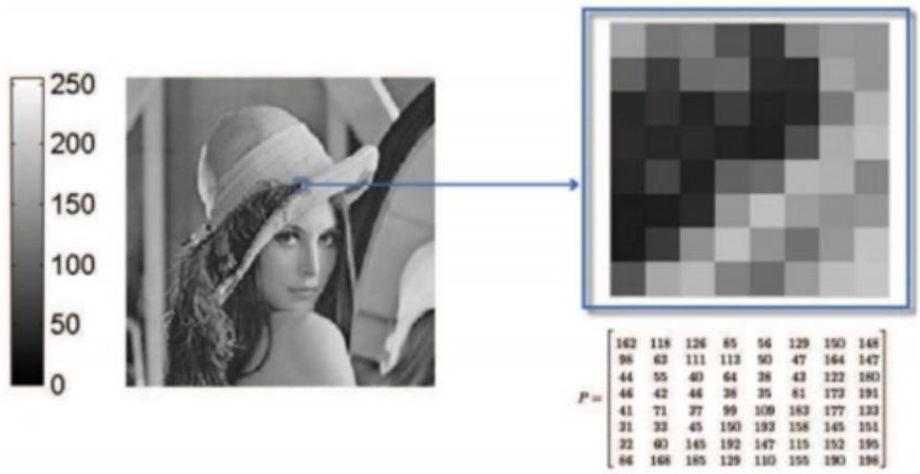
OpenCV je biblioteka otvorenog koda koja se pokazala kao popularan izbor knjižnice programskih funkcija za različite zadatke računalnog vida, pa tako i za detekciju tablica i klasifikaciju znakova. Široki spektar funkcija omogućuje veliku raznolikost manipulacije slikama i prilagodljivost algoritama specifičnim potrebama korisnika. U nastavku ćemo najprije objasniti što je računalni vid i čime se bavi, te zadatke detekcije i klasifikacije, odnosno raspoznavanja objekata. Na praktičnom primjeru pokazat ćemo realizaciju zadataka detekcije registarskih tablica i klasifikacije znamenaka na tablicama koristeći različite algoritme.

Računalni vid

Računalni vid je područje umjetne inteligencije koje omogućuje računalima i sustavima da zaključe smislene informacije iz digitalnih slika, videa i drugih vizualnih unosa [1]. Jedan od primjera korištenja računalnog vida je detekcija različitih vrsta objekata od kojih je jedan primjer prepoznavanje ljudskih lica. Prepoznavanje ljudskog lica se pokazalo korisnim u mnogim komercijalnim sektorima, a među najvažnijima je njegova uporaba u sektoru javne sigurnosti. Kina prednjači među državama u korištenju tehnologije prepoznavanja ljudskih lica i koristi ju između ostalog za policijske poslove, kod kontrolnih točaka u zračnim lukama i za sprječavanje krađa papira u Tiantan parku [23].

Rani eksperimenti s računalnim vidom su započeli još u 1950-ima dok je prvi put stavljen u komercijalnu uporabu u 1970-ima kad su pomoću njega krenuli tražiti razlike između tipkanog i pisanog teksta [2]. Interes u područje računalnog vida, te uspješnost programa iz tog područja doživljavaju ubrzani rast u današnje vrijeme a povezano je s napretkom tehnologije, povećanjem procesorske snage računala i kapaciteta prostora za pohranu, dostupnosti i rasprostranjenosti kamera na različitim uređajima, olakšanom razmjenom podataka putem Interneta i slično.

Smatra se da računalni vid procesira podatke na isti način kao i ljudski mozak, iako ne možemo u to biti sigurni [2]. Sva istraživanja o funkcioniranju ljudskog mozga su zasad zaključena isključivo teorijama te ne možemo definitivno zaključiti da li je to važeća usporedba ali ipak postoje podudarnosti: npr, ljudsko oko odgovara senzorima kamere, rezultat percepcije tj. fotografija ili vido snimljen kamerom se obrađuju, u slučaju ljudskog vida u mozgu u centrima za vid, a kod računalnog vida u računalu različitim algoritmima računalnog vida. Rezultat u oba slučaja je neki oblik interpretacije slike ili percipirane stvarnosti.



Slika 1. Primjer računalnog predstavljanja slike

Računalni vid uči računala o pozicijama objekata na slici, njihovom kretanju i o tome kojoj klasi pripadaju određeni objekti [1]. Da bi računalo prepoznalo neke uzorke ili objekte, potrebno je trenirati računalne metode s velikim brojem primjera istog objekta, na različitim udaljenostima od kamere, iz različitih kutova, s različitom pozadinom, s različitim varijacijama u kojima je moguće pojavljivanje tog objekta u realnom svijetu. Čim više slika se unese u proces treninga to će bolja biti uspješnost prepoznavanja tih objekata na novim slikama.

Detekcija objekata je zadatak računalnog vida koji uključuje klasifikaciju objekta i određivanje položaja objekata na digitalnim slikama i videima [24]. Primjeri algoritama opisani u ovom radu su specifični primjeri detekcije koji se odnose na detekciju registarskih tablica. Klasifikacija slike, odnosno objekata na slici je jedan od zadataka računalnog vida koji je zadužen za predviđanje vrste objekta u slici, tj. određivanje kojoj klasi pripadaju slika ili objekti u njoj. Procesom klasifikacije grupiramo objekte u unaprijed poznate kategorije na temelju njihovih sličnosti, [25], pa npr. na slici možemo imati prikazano više lica različitih osoba, a nakon klasifikacije program će zaključiti da su sva ta različita lica, ljudska lica. Da bi računala mogla ispravno klasificirati slike ili objekte na slikama koriste se metodama raspoznavanja uzoraka. Ovaj rad će proučiti primjer raspoznavanja znamenaka na registarskim tablicama.

OpenCV

OpenCV je ogromna knjižnica otvorenog koda (eng. open-source) za računalni vid, strojno učenje i obradu slika [3]. Inicijalno lansirana kao projekt Intel Research inicijative 1999., dobila je svoju Alpha verziju 2000. godine i pet Beta verzija krozsljedećih pet godina. Prva 1.0 verzija je lansirana 2006. godine. Sastoji se od C, C++,Python, Java i MATLAB sučelja i podržava Windows, Linux, Mac OS, iOS i Android [3][4].

OpenCV je pisan u C++ i zbog toga mu je C++ primarno sučelje. Iz tog razloga će se i u ovom radu koristiti C++. Verzija OpenCV-a koja će se koristiti je 4.5.3., a kodovi će se pisati i te testirati u Visual Studio Community 2019.

OpenCV je besplatan za korištenje što u poslovne, što u edukacijske svrhe. Potrebno je samo nekoliko jednostavnih koraka koji uključuju skidanje knjižnice i uključivanja ispravnih direktorija u svojstva konfiguracije Visual Studio-a da se krene sa korištenjem.

Haar kaskade

Haar kaskada klasifikator je program strojnog učenja za detekciju objekata koji identificira objekte u slici ili videu [5]. Primarno je korišten za detekciju lica, a temelji se na značajkama koje su dobivene detekcijom rubova ili linija [6]. Paul Viola i Michael Jones su ga prvi predložili u svojem istraživačkom radu imena ‘’Brzo otkrivanje objekata pomoću pojačane kaskade jednostavnih značajki’’ [7].

Prikazat ćemo primjer algoritma koji koristi Kaskadnu funkciju istreniranu na temelju slika ruskih registarskih tablica.

```cpp
#include <opencv2/imgcodecs.hpp>
#include <opencv2/highgui.hpp>
#include <opencv2/imgproc.hpp>
#include <opencv2/objdetect.hpp>
#include <iostream>

using namespace cv;
using namespace std;

int main() {

    string path = "Resources/002.jpg";
    Mat img = imread(path);
    resize(img, img, Size(800, 600), INTER_LINEAR);

    CascadeClassifier plateCascade;
    plateCascade.load("Resources/cascaderussian.xml");
    if (plateCascade.empty()) { cout << "XML file not loaded" << endl; }

    vector<Rect> plates;

    plateCascade.detectMultiScale(img, plates, 1.1, 10);

    for (int i = 0; i < plates.size(); i++)
    {
        rectangle(img, plates[i].tl(), plates[i].br(), Scalar(255, 0, 255), 3);

    }

    imshow("Image", img);
    waitKey(0);
    return 0;
}
```

Slika 2. Cjelovita .cpp datoteka potrebna za detekciju tablica na slici

Na slici iznad (Slika 2.) možemo primijetiti da sama .cpp datoteka potrebna za pokretanje programa za ispravnu detekciju se sastoji od vrlo malo linija koda. U ovom slučaju većina potrebnog koda se nalazi u unaprijed generiranoj *cascaderussian.xml* datoteci koja je istrenirana na negativnim i pozitivnim slikama da prepoznaje ruske registarske tablice. Sama .xml datoteka se sastoji od preko 2500 linija koda.

Slika 3. Primjer uspješno očitanih ruskih tablice preko algoritma Haar kaskada

Algoritam se pokazao veoma uspješan na prepoznavanju ruskih tablica kao što je prikazano i na primjeru iznad (Slika 3.). Nakon uspješnih rezultata na ruskim tablicama, algoritam smo testirali na devetnaest testnih slika, koje nisu bile skinute sa interneta nego fotografirane sa mobitelom i prenesene u originalnom stanju na računalo. One su poslužile kao glavna usporedba uspješnosti svih algoritama.

Rezultati su imali drastičan pad u uspješnosti naspram rezultata nad ruskim tablicama, ali i naspram drugim testnim tablicama koje su se detektirale sa slika pronađenih preko interneta. Usprkos tome, program je doprinio nekoliko točno očitanih rezultata čak i na ovome setu slika kao što je i vidljivo na slikama ispod (Slika 4. i 5.).

Slika 4. Primjer točno otkrivene hrvatske tablice preko Haar kaskada

Slika 5. Primjer točno otkrivene slovenske tablice preko Haar kaskada

Detekcija konturama

Koristeći detekciju kontura možemo otkriti rubove objekata i lokalizirati ih u slici. Kada spojimo sve vrhove na granici objekta, dobijemo konturu [8]. Detekciju kontura smo koristili u sva preostala tri algoritma. Sami algoritmi se razlikuju u tome koje filtere koriste, te koje konture traže i kako ih traže. Traženje kontura pomoću OpenCv-a je uvelike olakšano, te zahtjeva korištenje samo dviju funkcija, findContours() i drawContours().

Canny detektor rubova

Canny detektor rubova [9] je operator za detekciju rubova koji se sastoji od tri promjenjiva parametra: širine Gaussove krivulje (što je slika bučnija to je širina veća), te donjeg i gornjeg praga za prag histereze. Proces rada Canny algoritma se može svesti na pet koraka:

1. Nanese Gaussov filter za uglađivanje slike kako bi uklonio šum
2. Pronađe gradijente intenziteta slike
3. Primjeni prag veličine gradijenta ili potiskivanje granične vrijednosti donje granice kako bi se riješio lažnog odgovora na otkrivanje rubova
4. Primjeni dvostruki praga za određivanje potencijalnih rubova
5. Praćenje rubova histerezom: Završi otkrivanje rubova izbacivanjem svih ostalih slabih rubova koji su nastali zbog šuma i nisu povezani s jakim rubovima

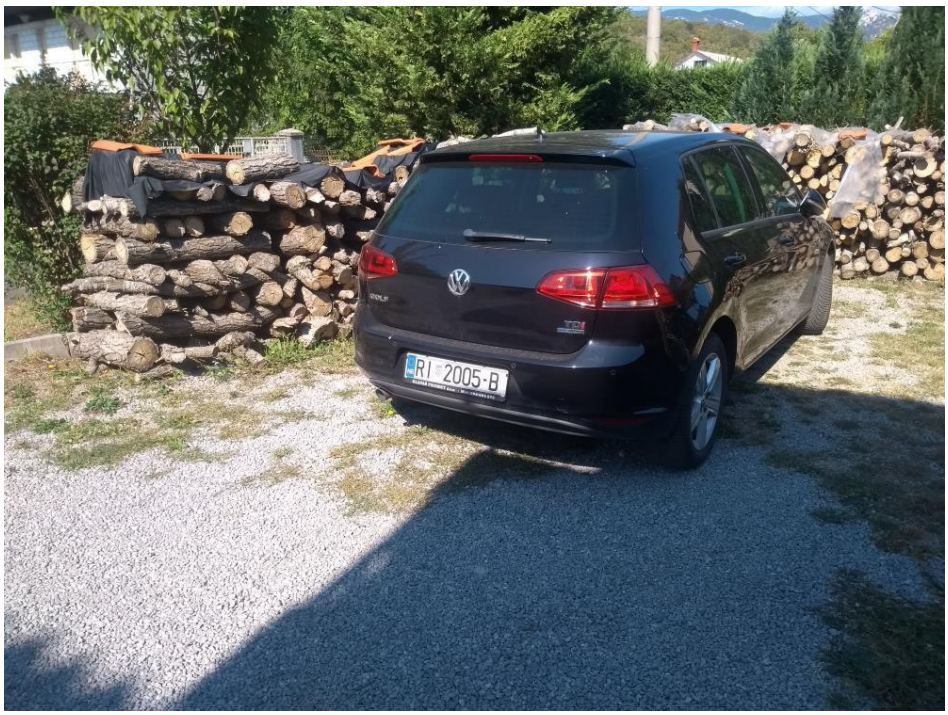
Pronađeni primjer Canny algoritma je bio napisan u Pythonu. Za održavanje jednakosti među algoritmima, u slučaju ako se pokaže da Python ima prednost ispred C++ ili zaostaje za njim kao optimalan programski jezik za detekciju objekata na slici, u ovom radu koristiti ćemo vlastiti Canny algoritam napisan na temelju pronađenog Pythonovog koda.

```
string path = "resources/001.jpg";
Mat img = imread(path);
//tretiranje slike
resize(img, img, Size(600, 400), INTER_LINEAR);

Mat greyMat;
cvtColor(img, greyMat, COLOR_BGR2GRAY);

Mat filtered;
bilateralFilter(greyMat, filtered, 13, 20, 20);

Mat afterCanny;
Canny(filtered, afterCanny, 30, 200);
```

Slika 6. Učitavanje slike i primjenjivanje filtera potrebnih za Canny algoritam

Na slici iznad (Slika 6.) možemo vidjeti početni dio koda koji koristi Canny algoritam. Najprije uz pomoć resize() funkcije smo sve slike sveli na istu veličinu. U konačnici ovaj korak je bio ponajviše bitan za konačni prikaz slike. Sama slika se mogla obraditi u bilo kojoj veličini. Zadržali smo resize() na početku jer slika koja se obrađuje i konačna slika su morale biti iste veličine da bi se kontura koja se pronađe na obrađenoj slici prikaže na ispravnom mjestu na konačnoj slici. Nakon toga smo pomoću funkcije cvtColor() sliku iscrtali u grayscaleu: Verziji slike gdje vrijednost svakog pixela je jedan uzorak koji predstavlja količinu svjetlosti. Ovaj korak je potreban da bi Canny() funkcija mogla ispravno očitavati rubove. Koristili smo bilateralFilter umjesto Gaussovog jer u primjeru na temelju kojeg je ovaj kod napisan se također koristio bilateralFilter i zbog toga što ćemo vidjeti primjenu Gaussovog filtera u sljedeća dva programa. Jedini parametri koji su promijenjeni u odnosu na zadani Python primjer su *sigmaColor* i *sigmaSpace* unutar bilateralnog filtera. Podignuti su sa 15 na 20 u pokušaju da se zamuti više pozadine i time izbaci čim više nebitnih rubova.

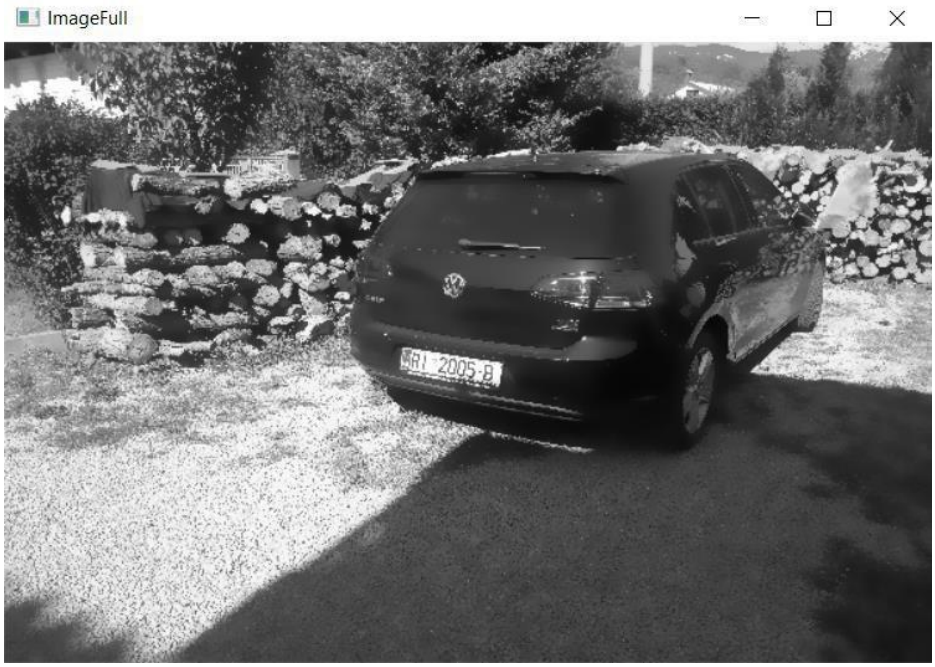
Tijekom testiranja su se također pomicale i granice pragova unutar Canny() funkcije ali nije bilo pozitivnih pomaka. Naprotiv, previsok donji prag i prenizak gornji prag su bili doveli do gubljenja bitnih rubova.



Slika 7. Originalna slika

Slika (Slika 7.) iznad je prva testna slika u početnom stanju. Ispod možemo vidjeti kako ta slika izgleda nakon tretiranja funkcijom cvtColor() (Slika 8.), bilateralnim filterom (Slika 9.) i na kraju Canny() funkcijom (Slika 10.). Možemo vidjeti da su pronađeni mnogi rubovi koji će nam potencijalno smetati u detekciji, ali među njima možemo vidjeti i rubove koji tvore konturu oko tablice.

Slika 8. Grayscaled slika poslije cvtColor() funkcije



Slika 9. Slika nakon primjene bilateralFilter() funkcije

Slika 10. Slika nakon primjene Canny() funkcije

Nakon primjene Canny algoritma za detekciju rubova slika je spremna za pronaći konture koje ti rubovi tvore. Koristimo ranije napomenutu funkciju findContours() koja pronađe sve vidljive konture na slici i spremi ih u naš dvodimenzionalni vektor (Slika 11.). Te konture nakon toga posložimo po veličini (Slika 11. i 12.) i izdvojimo 10 najvećih. Pretpostavka je da će naša kontura tablice uvijek biti u 10 najvećih.

```
vector<vector<Point>> contours;
vector<Vec4i> hierarchy;
findContours(afterCanny, contours, hierarchy, RETR_TREE, CHAIN_APPROX_SIMPLE);

sort(contours.begin(), contours.end(), compareContourAreas);
for (int i = 1; i <= contours.size(); i++) {
    vector<Point> currentContour = contours[contours.size() - i];
    if (i > 10) {
        contours.erase(contours.end() - i + 1);
        i--;
    }
}
```

Slika 11. Dio koda za pronalazak 10 najvećih kontura

```
bool compareContourAreas(vector<Point> contour1, vector<Point> contour2) {
    double i = fabs(contourArea(Mat(contour1)));
    double j = fabs(contourArea(Mat(contour2)));
    return (i < j);
}
```

Slika 12. Funkcija za usporedbu veličina kontura

```
vector<Point> screenCnt;
vector<Point> approx;
int detected;
int zfound;

for (int z = 0; z <= contours.size(); z++) {

        approxPolyDP(contours[z], approx, arcLength(contours[z], true) * 0.018, true);
        if (approx.size() == 4) {
            zfound = z;
            screenCnt = approx;
            break;
        }
}

if (screenCnt == approx) {
    detected = 1;
    cout << "CONTOUR DETECTED";
}
else {
    detected = 0;
    cout << "No contour detected";
    return 0;
}


if (detected == 1) {
    //for (int j = 0; j < contours.size(); j++) {
        Scalar color = cv::Scalar(rng.uniform(0, 255), rng.uniform(0, 255), rng.uniform(0, 255));
        drawContours(img, contours, zfound, color, 3);
    //}
}
```

Slika 13. For petlja za pronalazak željene konture i drawContours za iscrtati ju

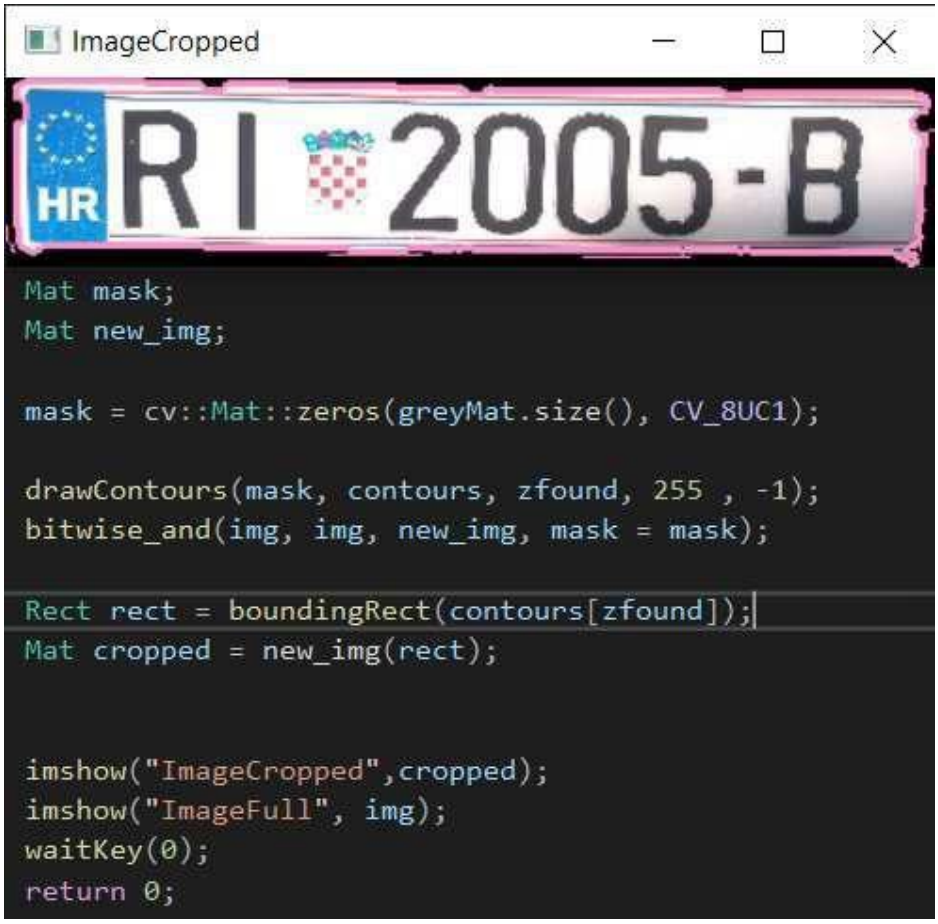
Na slici iznad (Slika 13.) možemo vidjeti da pokušava naći željenu konturu uz pomoć approxPolyDP() funkcije koja približno odredi oblik željene konture, usporedi sa dobivenom konturom i spremi u vektor approx koji onda testira da li ima točno 4 ruba kao što bi pravokutnik trebao imati. U slučaju da ima smatramo da je to naša tablica i uz pomoć drawContours() funkcije ju nacrtamo. Stavljanjem *–1* umjesto *zfound* unutar drawContours() funkcije možemo iscrtati svih 10 kontura da najprije provjerimo je li kontura tablice uopće bila među tih 10 koje smo promatrali. Na slici ispod (Slika 14.) možemo vidjeti dobivene rezultate kad smo stavili *j* umjesto *zfound* i otkomentirali for petlju tako da svaka od 10 prikazanih kontura bude jasno vidljiva drugom bojom od ostalih.

Slika 14. Iscrtanih svih 10 najvećih kontura koje je program našao na ovoj slici

Možemo vidjeti da kontura koja nas zanima, ona koju tvore rubovi registarske tablice, uopće nije među preostalima (Slika 14.). Zbog toga što se ne nalazi među njima znamo da će rezultat biti kriva kontura ili da neće očitati nijednu konturu kao točnu i napisati ‘’No contour detected’’ u command lineu. Daljnje testiranje je pokazalo da je zaista došlo do slučaja gdje je program odabrao krivu konturu i smatrao je da je raslinje u pozadini označeno ljubičastim rubovima tražena kontura (Slika 14.). Da bi provjerili koji dio koda je odgovoran za krivo dobivenu konturu zakomentirali smo dio koji briše sve manje konture i pustili da se for petlja sa slike 12. provede nad svim pronađenim konturama na slici. Rezultat je bio različit i program je vratio konturu koja nije bila među deset najvećih, ali je opet bio kriv. Vraćena kontura je bila mali uzorak među kamenčićima, a ne tražena tablica. Od devetnaest testnih slika Canny algoritam je vratio jedan točan (Slika 14.) i 18 krivih rezultata, uvijek pronalazeći konturu za koju je program smatrao da je točna po traženim parametrima.

Slika 15. Jedini točan rezultat Canny algoritma

Veoma niska razina uspješnosti ovog algoritma pokazuje na to da zahtjeva mnoga dotjeravanja prije nego što postane koristan. Testne slike sa ostalih izvora od vlastitih su pokazale veći stupanj uspješnosti ali još uvijek nedovoljno visok da zadovolji potrebe programa za prepoznavanje. Veći stupanj rezolucije vlastitih slika je najizgledniji krivac za razliku u uspješnosti. Slike iz drugih izvora su hvatale puno manje irelevantnih kontura, tako da je program imao veće šanse za pronalazak ispravne. Kao što se vidina slici iznad (Slika 15.), koja je jedina vratila točan rezultat, fokusiranje na tablicu izbliza na samoj originalnoj fotografiji je pomoglo programu na sličan način kako su pomogle i ostale slike koje je točno detektirao. Manje neravnih površina i rubova značilo je da je program učitao manje kontura i s time otpočetka imao veće šanse za ispravno prepoznavanje. Smatram da bi za povećanu uspješnost ovog programa ponajprije trebalo uvesti više parametara koji bi određivali je li vektor *approx* zaista kontura koju tražimo ili ne. Na slici ispod (Slika 16.) se pronalazi još zadnji dio koda programa koji nakon pronalaska tražene konture napravi masku kako bi zacrnili nepotrebne podatke izvan same konture i nakon toga izrežemo konturu u novi prozor preko boundingRect() funkcije.



Slika 16. Posljedni dio koda kod Canny algoritma sa povratnom slikom izrezane tablice

Na slici iznad (Slika 16.) je dakle prikazano stvaranje maske tako što funkcijom zeros() ispunimo novi Mat *mask* s onoliko nula koliko ima pixela u originalnoj grayscaled slici da dobijemo novu potpuno crnu sliku koja je iste veličine kao naša slika. Naravno sve varijable koje sadržavaju slike su tipa Mat jer to je naš osnovni spremnik slika koji nam omogućuje da s tim slikama onda radimo kao s podacima. Nakon toga opet nacrtamo konturu na pronađenom mjestu ali ovaj put umjesto crtanja ruba ispunimo cijelu unutrašnjost sa bijelom bojom. Pomoću bitwise_and() funkcije stvorimo novu sliku koja umjesto bijele pozadine prikaže dio slike koji se tamo nalazi na našoj rezultantnoj slici, a ostatak nove slike pusti crnom.

Sliku zatim izrežemo kako bi lakše na njoj primijenili algoritam za prepoznavanje znakova. Ovaj program ipak nema taj dio koda jer se nije pokazao kao dovoljno dobar izbor da ga se implementira u ovom trenutku.

Određivanje praga slike

Preostali dva algoritma koje ćemo testirati umjesto Canny-a koriste definirani prag (eng. thresholding) za pronalazak mogućih kontura na slici. Određivanjem praga na najjednostavniji način se mogu segmentirati slike u području digitalne obrade slike [19]. Kao i Canny implementira se na grayscale verziju slika. Pomoću praga se od grayscale verzija dobiju binarne slike na kojima se tada mogu tražiti konture. Algoritmi koji su preostali za testiranje u ovom radu se razlikuju u tome što jedan dopunjuje jednostavno određivanje praga (eng. simple thresholding) sa traženjem bijelih regija dok drugi koristi prilagodljivo određivanje praga (eng. adaptive thresholding). Kod jednostavnog određivanja praga princip je slijedeći: ako se vrijednost pixela nalazi iznad praga, on poprima vrijednost 1, a ako se nalazi ispod praga 0. Kod prilagodljivog praga algoritam mijenja vrijednost praga ovisno o osvjetljenju područja [18].

Jednostavno određivanje praga

Algoritam jednostavnog određivanja praga koji možemo vidjeti na slici ispod (Slika 17.) provodi samu funkciju threshold() dva puta. Jednom na početnoj slici da pronađe bijele regije i jednom na slici koja je tretirana Gaussovim filterom. Te slike tada preklopi u jednu da dobije čim preciznije prikazane moguće kandidate za traženu konturu.

```
// Perform blackhat morphological operation, reveal dark regions on light backgrounds
cv::Mat blackhatFrame;
cv::Mat rectangleKernel = cv::getStructuringElement(cv::MORPH_RECT, cv::Size(13, 5));
// Shapes are set 13 pixels wide by 5 pixels tall
cv::morphologyEx(processedFrame, blackhatFrame, cv::MORPH_BLACKHAT, rectangleKernel);

// Find license plate based on whiteness property
cv::Mat lightFrame;
cv::Mat squareKernel = cv::getStructuringElement(cv::MORPH_RECT, cv::Size(3, 3));
cv::morphologyEx(processedFrame, lightFrame, cv::MORPH_CLOSE, squareKernel);
cv::threshold(lightFrame, lightFrame, 0, 255, cv::THRESH_OTSU);

// Compute Sobel gradient representation from blackhat using 32 float,
// and then convert it back to normal [0, 255]
cv::Mat gradX;
double minVal, maxVal;
int dx = 1, dy = 0, ddepth = CV_32F, ksize = -1;
cv::Sobel(blackhatFrame, gradX, ddepth, dx, dy, ksize);
// Looks coarse if imshow, because the range is high?
gradX = cv::abs(gradX);
cv::minMaxLoc(gradX, &minVal, &maxVal);
gradX = 255 * ((gradX - minVal) / (maxVal - minVal));
gradX.convertTo(gradX, CV_8U);

// Blur the gradient result, and apply closing operation
cv::GaussianBlur(gradX, gradX, cv::Size(5, 5), 0);
cv::morphologyEx(gradX, gradX, cv::MORPH_CLOSE, rectangleKernel);
cv::threshold(gradX, gradX, 0, 255, cv::THRESH_OTSU);

// Erode and dilate
cv::erode(gradX, gradX, 2);
cv::dilate(gradX, gradX, 2);

// Bitwise AND between threshold result and light regions
cv::bitwise_and(gradX, gradX, lightFrame);
cv::dilate(gradX, gradX, 2);
cv::erode(gradX, gradX, 1);
```

Slika 17. Algoritam jednostavnog određivanja praga

Slika 18. Primjer djelomično točne detekcije

Na primjeru iznad (Slika 18.) možemo vidjeti jedan od rezultata dobivenih preko algoritma za jednostavno određivanje praga. Konačna slika nam je dala djelomično točan rezultat gdje je tablica točno označena kao potencijalni kandidat ali je također označeno i područje na jednom od nosača. Područje se naizgled ističe kao veoma svijetli dio slike i promatrajući ostale rezultate kod ovog algoritma kod kojih smo dobili djelomično točne ili krive rezultate može se primijetiti da je algoritam imao najviše problema u slučajevima kad su na slici postojali objekti podjednake ili veće svjetline.

Prilagodljivo određivanje praga

Na slici ispod (Slika 19.) vidimo da algoritam prilagodljivog određivanja praga također koristi Gaussov filter za zamućivanje slike i da slika na kojoj koristi taj filter također mora biti u grayscale-u.

```
void preprocess(cv::Mat &imgOriginal, cv::Mat &imgGrayscale, cv::Mat &imgThresh) {
    imgGrayscale = extractValue(imgOriginal);                       // extract value channel only from original image to get imgGrayscale

    cv::Mat imgMaxContrastGrayscale = maximizeContrast(imgGrayscale);       // maximize contrast with top hat and black hat

    cv::Mat imgBlurred;

    cv::GaussianBlur(imgMaxContrastGrayscale, imgBlurred, GAUSSIAN_SMOOTH_FILTER_SIZE, 0);          // gaussian blur

                                                                                                    // call adaptive threshold to get imgThresh
    cv::adaptiveThreshold(imgBlurred, imgThresh, 255.0, cv::ADAPTIVE_THRESH_GAUSSIAN_C, cv::THRESH_BINARY_INV, ADAPTIVE_THRESH_BLOCK_SIZE, ADAPTIVE_THRESH_WEIGHT);
}
```

Slika 19. Algoritam prilagodljivog određivanja praga

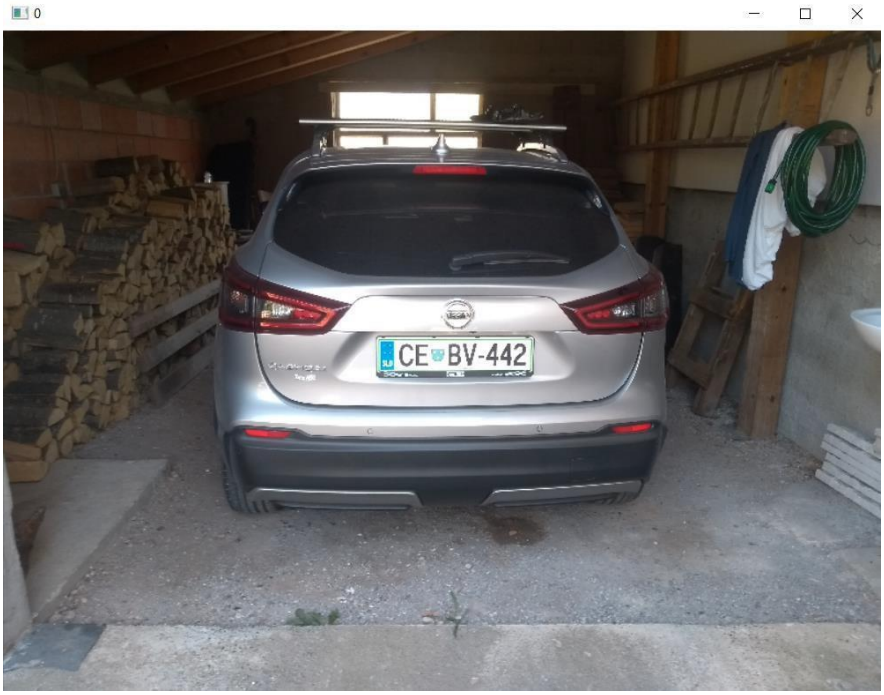
Ovaj program se daljnje razlikuje od ostalih što odmah traži konture mogućih znakova tablice umjesto samu konturu ruba tablice.

Zbog najvećeg stupnja uspješnosti ovog algoritma kod detekcije tablica, iskorišten je i za testiranje prepoznavanja znakova sa dobivenih tablica.

Dodatno, nakon što se odredi dio slike na koje se nalazi registarska tablica koristi se k-NN algoritam za raspoznavanje znakova na tablici.

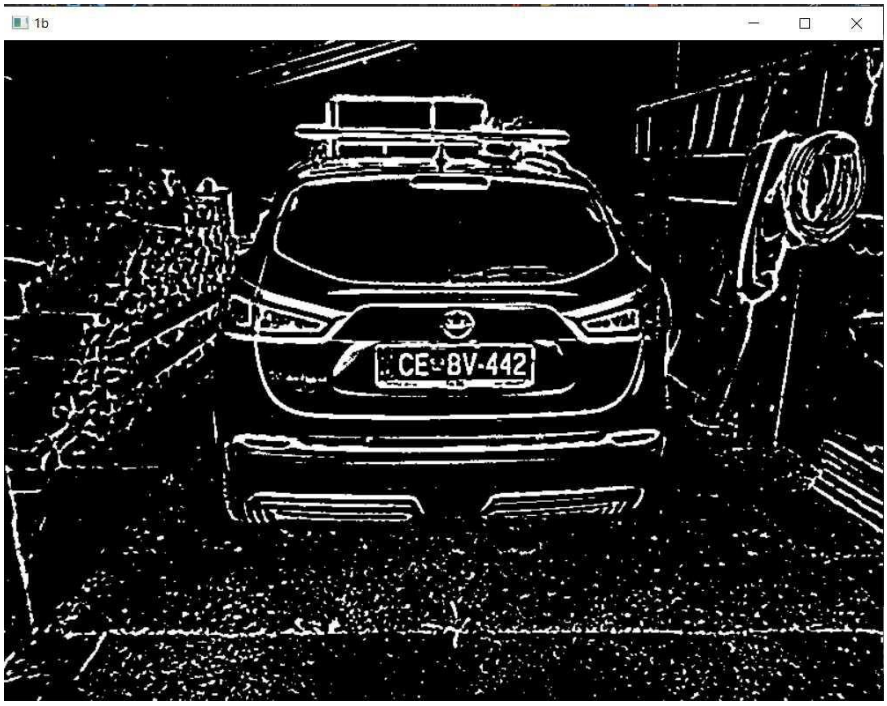
K-NN algoritam

K-NN algoritam ili algoritam k-najbližih susjeda je jednostavan i lagan za implementaciju. Služi za nadzirano strojno učenje i može poslužiti za rješavanje problema klasifikacije i regresije [22]. Kod našeg primjera uveli smo ga za rješavanje problema klasifikacije, preciznije raspoznavanje znakova koji se mogu pojaviti na registarskim tablicama. Na sljedećim slikama pokazati ćemo za primjer sve korake kroz koje prolazi jedna slika u našem programu, od pripreme za određivanje praga preko traženja kontura do krajnjih rezultata.



Slika 20. Slika učitana u originalnom stanju

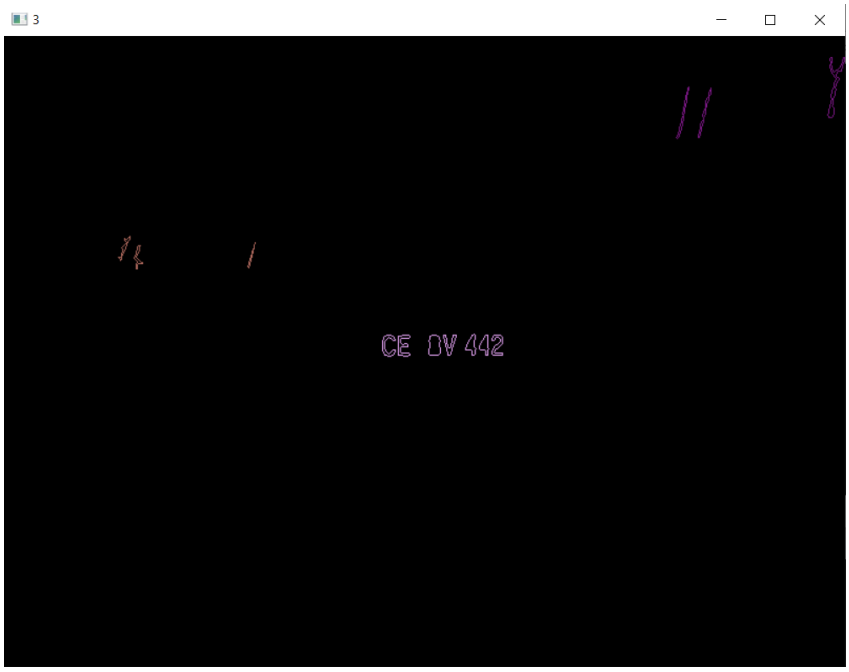
Slika 21. Grayscale verzija slike



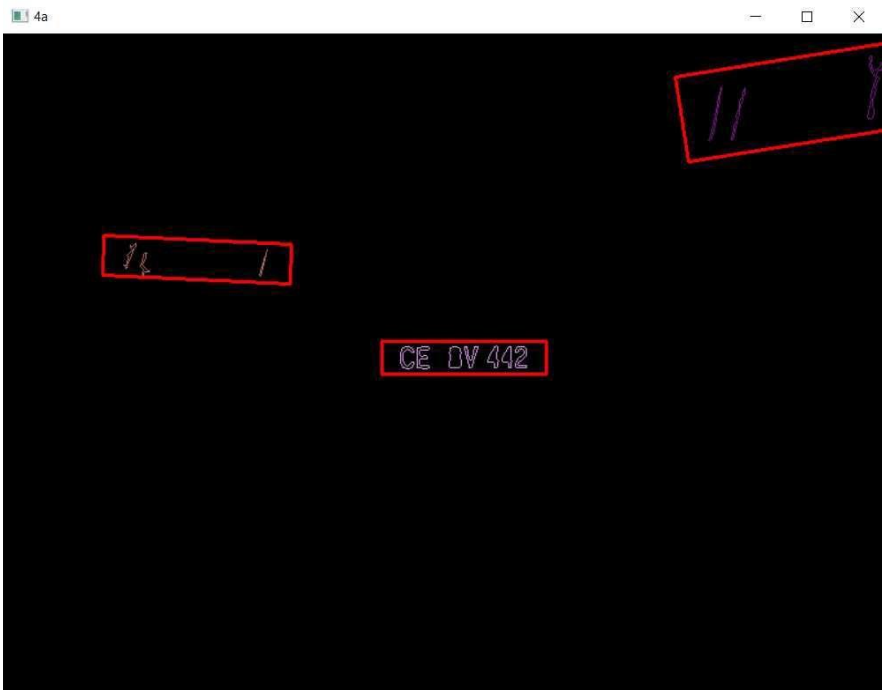
Slika 22. Slika nakon provedenog određivanja praga

Slika 23. Slika sa svim pronađenim konturama

Slika 24. Sa slike su izdvojene samo konture koje program prepoznaje kao moguće znakove



Slika 25. Među svim potencijalnim znakovima izdvojene su potencijalne grupe znakova

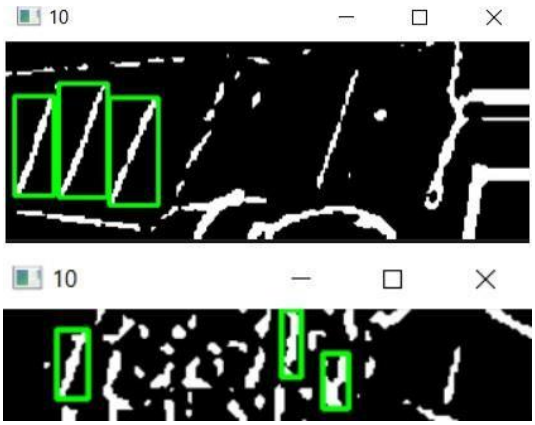


Slika 26. Program izdvaja jednu po jednu sve grupe znakova da bi očitao znakove

Na svakoj izrezanoj slici od grupa znakova program zatim izvodi iste korake koje je proveo nad prvotnom slikom da bi dobio jasno čitljive konture. Konture su ovaj put zasebno iščitane da bi se dobio po jedan znak iz svake.

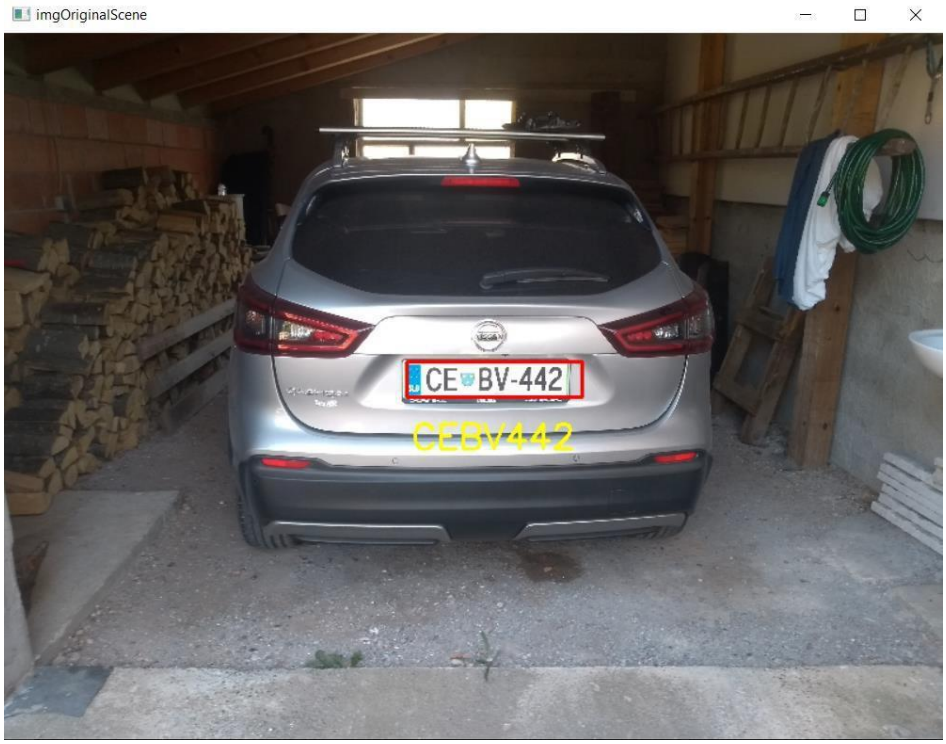
Slika 27. Određivanje praga novih slika

Slika 28. Daljnja obrada slika da bi se dobile točne konture i izvukli znakovi



Slika 29. Rezultati ostalih mogućih skupina znakova

Nakon što smo na slikama iznad (Slike 20.-29.) mogli vidjeti detaljan proces programa i kako dolazi do željenih rezultata, sada možemo na slici ispod (Slika 30.) pogledati krajnji rezultat programa. Vidimo da je program u ovom slučaju vratio u potpunosti točno očitavanje znakova tablice.



Slika 30. Krajnji rezultat program

Preostale dvije skupine mogućih znakova je očitao ali zbog kraće duljine liste podudarajućih znakova ih je zanemario kao mogući točan rezultat. Proces također možemo pratiti i preko command line-a, prikazan na slikama ispod (Slika 31. i Slika 32.).

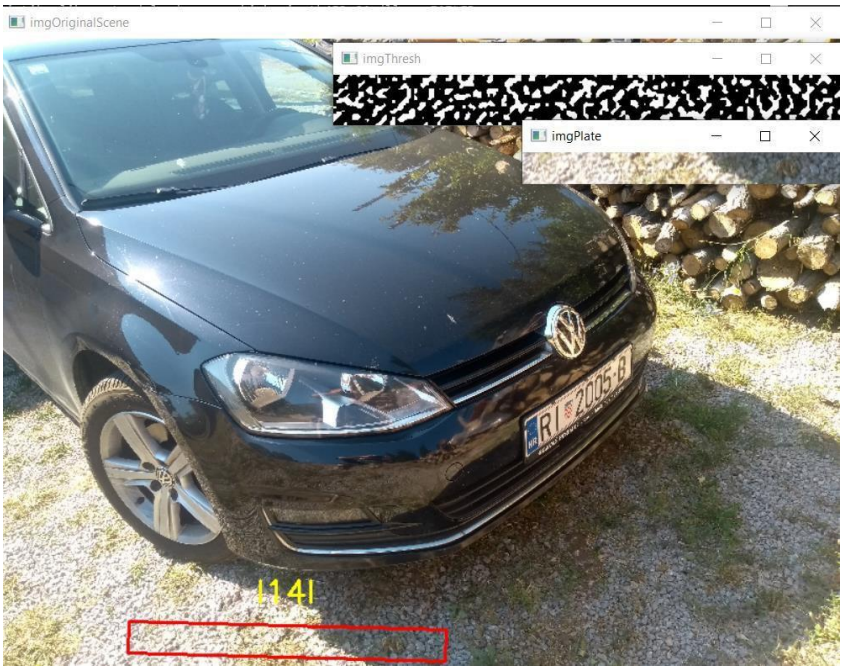
```
contours.size() = 2061
step 2 - intCountOfValidPossibleChars = 64

step 2 - vectorOfPossibleCharsInScene.Count = 64

step 3 - vectorOfVectorsOfMatchingCharsInScene.size() = 3


3 possible plates found
```

Slika 31. Proces izdvajanja kontura prikazan u command line-u

```
chars found in plate number 0 = CEBV442, click on any image and press a key to continue . . .
chars found in plate number 1 = 770, click on any image and press a key to continue . . .
chars found in plate number 2 = 77X, click on any image and press a key to continue . . .

char detection complete, click on any image and press a key to continue . . .
```

Slika 32. Liste znakova sa očitanih mogućih područja tablice

Kod ostalih primjera program je imao najviše problema sa slikama koje su slikane iz većeg kuta, a ne ravno iza tablice kao što možemo vidjeti na primjeru ispod (Slika 33.).



Slika 33. Primjer krivo detektirane tablice

Problem slabog očitavanja tablica kod fotografija slikanih iz kuta bi se potencijalno mogao riješiti tako da dodatno istreniramo program preko kojeg smo dobili .xml datoteke sa primjerima znakova slikanih iz kuta. Tada bi mogli zamijeniti novodobivene .xml datoteke s našim trenutnim za poboljšanu preciznost očitavanja znakova koji su prikazani pod kutem. Program je također naišao na probleme kod očitavanja slova ‘’I’’ čak i kada bi sve ostalo točno očitao (Slika 34.). Nakon pregledavanja dokumentacije koda koji je poslužio za treniranje pregledavanja znakova može se pretpostaviti da razlog leži u tome što je istrenirano na setu znakova koji koriste Sjedinjene Američke Države u svojim registarskim tablicama. Za razliku od hrvatskih, američke tablice pišu svoje ‘’I’’ sa dodatnim linijama na donjem i gornjemvrhu.

Slika 34. Skoro u potpunosti točno očitana tablica

Usporedba rezultata

Sljedeća tablica rezultata (Slika 35.) odnosi se samo na detekciju tablica pošto je prepoznavanje znakova primijenjeno samo na najuspješniji od četiri algoritma.

		CONTOURS		
REZULTATI	HAAR CASCADES	CANNY	LIGHT REGIONS + THRESHOLD	ADAPTIVE THRESHOLD
KRIVO	0	18	11	8
NEMA REZULTATA	15	0	5	3
DJELOMICNO TOCNO	0	0	2	1
TOCNO	4	1	1	7

Slika 35. Tablica usporedbe konačnih rezultata detekcije

Što se tiče samog raspoznavanja znakova, od sedam točnih detekcija algoritam sa prilagodljivim pragom je vratio dvije u potpunosti točno raspoznate tablice, dvije gdje mu je nedostajao jedan znak, jednu gdje je imao jedan znak viška, jednu gdje je imao jedan znak viška i jedan manjka i jednu gdje je imao jedan znak manjka i jedan znak je krivo očitao. Djelomično točna detekcija je vratila četiri točno očitana znaka, jedan kojeg nema na tablici i nije uspjela očitati preostala tri.

Zaključak

Detekcija registarskih tablica i raspoznavanje znakova kompleksno područje koje ima nekoliko potencijalnih veoma korisnih upotreba na cesti i u provođenju zakona. Načini na koje možemo pristupiti potrazi za čim uspješnijim rezultatima su također raznovrsni kao što smo vidjeli i na našim primjerima algoritama.

Korišteni algoritmi su vratili veoma različite razine uspjeha detekcije tablica, ali na kraju čak ni najuspješniji, algoritam prilagodljivog praga, nije došao niti do 50% uspješnosti. Jedna činjenica je da su algoritmi pokazali daleko uspješnije rezultate kod slika koje su bile slikane iz kuta iz kojeg se jasno vidio obrub tablice, te očekivana širina i pozicija znakova. Dobro pozicionirana kamera bi mogla dovesti i do značajno boljih rezultata kod praktičnih primjena.

Pozitivna strana je što već sada postoji mnoštvo opcija koje bi mogle povećati uspješnost tih algoritama. Uz algoritam prilagodljivog određivanja praga, Haar kaskadni algoritam istreniran isključivo na ruskim tablicama se pokazao posebno zanimljivim te iako je imao manju uspješnost postoji mogućnost da bi mogao biti najuspješniji kada bi ga se istreniralo na hrvatskim tablicama.

Iz toga možemo zaključiti da ove testirane metode nisu dovoljne za komercijalnu primjenu te da čak niti raspoznavanje tablica ne možemo „hard kodirati“ jer i u malom broju primjera postoje velike razlike među tablicama na slici. Očito je da bi ove metode postale prisutnija u našim životima u nadolazećim godinama da ih je potrebno nadograditi metodama koje će moći bolje detektirati tablice i bolje raspoznati znakove na tablicama na neki drugi način, kao što je učeći na velikom broju primjera kao što to rade metode strojnog učenja i koristeći već naučene metode na sličnim slučajevima slika.

Prilozi

Uz rad priložene su sljedeće datoteke

- tablice.zip (Paket sadrži pet mapa, četiri sa svojim odgovarajućim datotekama koda, te jednu mapu u kojoj se nalaze testne fotografije)

Popis slika

Literatura i izvori

1 - https://www.ibm.com/topics/computer-vision

2 - https://towardsdatascience.com/everything-you-ever-wanted-to-know-about-computer-vision-heres-a-look-why-it-s-so-awesome-e8a58dfb641e

3 - https://www.geeksforgeeks.org/opencv-overview/

4 - https://opencv.org/about/

5 - https://medium.com/analytics-vidhya/haar-cascades-explained-38210e57970d

6 - https://towardsdatascience.com/face-detection-with-haar-cascade-727f68dafd08

7 - https://www.cs.cmu.edu/~efros/courses/LBMV07/Papers/viola-cvpr-01.pdf

8 - https://learnopencv.com/contour-detection-using-opencv-python-c/

9 - https://medium.com/@ssatyajitmaitra/what-canny-edge-detection-algorithm-is-all-about-103d94553d21

10 - https://en.wikipedia.org/wiki/Canny_edge_detector

11 - https://towardsdatascience.com/canny-edge-detection-step-by-step-in-python-computer-vision-b49c3a2d8123

12 - https://justin-liang.com/tutorials/canny/

13 - https://en.wikipedia.org/wiki/Grayscale

14 - https://en.wikipedia.org/wiki/OpenCV

15 - https://hr.wikipedia.org/wiki/Računalni_vid

16 - https://www.analyticsvidhya.com/blog/2018/03/introduction-k-neighbours-algorithm-clustering/

17 - https://towardsdatascience.com/computer-vision-detecting-objects-using-haar-cascade-classifier-4585472829a9

18 - https://opencv24-python-tutorials.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_thresholding/py_thresholding.html

19 - https://en.wikipedia.org/wiki/Thresholding_(image_processing)

20 - https://docs.opencv.org/3.4.14/d7/dd0/tutorial_js_thresholding.html

21 - https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm

22 - https://towardsdatascience.com/machine-learning-basics-with-the-k-nearest-neighbors-algorithm-6a6e71d01761

23 - https://www.forbes.com/sites/bernardmarr/2019/04/08/7-amazing-examples-of-computer-and-machine-vision-in-practice/

24 - https://machinelearningmastery.com/object-recognition-with-deep-learning/

25 - https://www.researchgate.net/publication/280580698_Image_Classification_and_Pattern_Recognition