Közlekedési táblák felismerése – dokumentáció

Név: Falatics Noémi

Neptun kód: DRQGPU

Bevezetés, megoldandó feladat kifejtése

A tantárgy teljesítéséhez olyan feladatot szerettem volna választani, ami a mai világban hasznos, és az érdeklődési körömbe is beletartozik. Mivel szeretek vezetni és érdekesnek találom az önvezetés témáját is, ezért a megadott témák közül a közlekedési táblák felismerése témát választottam.

A feladat minimum 5 közlekedési táblát felismerő program írása. A bemeneti kép alapján a program a terminálra kiírja, hogy milyen tábla található az adott képen. Amennyiben több tábla található a képen, azt a program lekezeli, tehát mindegyik tábla nevét kiírja. A program csak azokat a táblákat ismeri fel a képen, amelyeket előre meghatározok, amelyek a következők: Stop tábla, Elsőbbség adás tábla, Zsákutca tábla, Körforgalom tábla, Főútvonal tábla. A többi közlekedési táblát a program nem kezeli. A bemeneti képen a megtalált táblát bekeretezéssel jelöli a program, amit kimeneti képként kapunk meg.

Megoldáshoz szükséges elméleti háttér rövid ismertetése

A képfeldolgozás és a gépi látás egyik legfontosabb alapművelete az éldeteketálás, hiszen az élek általában a tárgyak határvonalánál találhatóak, így ismertethetőek fel a programmal a különböző tárgyak.

Az éldetektálást nehezíti, hogy élek nemcsak a tárgyak határvonalánál láthatóak, hanem például a tárgyak árnyékainál, vagy a három dimenziós tárgy felületén lévő törésvonalnál.

Él ott található egy képen, ahol az intenzitás értéke hirtelen nagyot ugrik vagy esik. Ennek következtében első körben az intenzitásokat egy függvényen ábrázoljuk, majd ezt a függvényt deriváljuk. A deriváltfüggvény megmutatja a változás nagyságát és irányát.

A képek, és így a belőlük készített intenzitásfüggvények széleinél nincs két érték, csak egy, így differenciálszámítást ott nem tudunk végezni, ezért a határelemeket kezelni kell. Ezen felül pedig az is nehezíti a problémát, hogy a képek két dimenziósak, így valójában parciális deriválást kell végrehajtani.

A képek deriváltját konvolúciós szűrőkkel szokták közelíteni. Az egy dimenziós szűrők érzékenyek a zajokra, így a gyakorlatban a két dimenziós szűrők terjedtek el, ilyen például a Prewitt vagy a Sobel.

A Laplace szűrő viszont már második deriválttal dolgozik, így még szebb eredmény várható.

A legelterjedtebb módszer azonban a Canny detektor, ami több lépésből áll: a kép szürkeárnyalatú koverziója után Gauss simítást végez, majd az éldetektálás alapját jelentő differencia számítás következik, ezután nem-maximum vágás, kettős küszöbölés és hiszterézis küszöbölés jön.

Az élek ilyen módszerekkel meghatározásra kerülnek.

A feladat megoldásához szükséges másik fontos elméleti rész a színek kezelése. A képek 3 színcsatornára tartalmaznak pixelenként értékeket 0 és 255 között. A színcsatornák a piros (red), zöld (green) és kék (blue) színek megfelelő mértékű “keveresével”, intenzitásával adják meg a pixel adott színét.

A képek kijelzésénél a számítógépek számára az RGB egy jól működő rendszer, az emberi látáshoz, színérzékeléshez azonban közelebb áll a HSV (színárnyalat (hue), telítettség (saturation), érték (value)). Ebben a modellben az egyes árnyalatok színei radiális szeletben vannak elrendezve, a semleges színek központi tengelye körül, amely alul feketétől a felső fehérig terjed. A HSV modellben az egyes jellemzők sorra 0-180, 0-255 és 0-255 között vehetnek fel értékeket.

A mintakeresés esetén az a jellemző, és a megoldandó feladatban is így van, hogy a minta nem arról a képről származik, amin keressük a mintát. Ez miatt a minta és a képrészlet nem fog 100%-osan egyezni. A mintakereső algoritmus azt a területet fogja megmutatni, ahol a keresett minta a kép adott területű pixeleivel a legjobban korrelál.

A másik probléma a mintakereséssel a kép és a minta mérete. A mintát érdemes lehet kisebb felbontású képen keresni először, hiszen nagy felbontású kép esetén rengeteg erőforrást használ a gép, hogy megvizsgálja a kép összes pixelét, mennyire egyezik meg a mintával.

Ez miatt a kicsinyítés és nagyítás problémaköre is olyan elméleti rész, aminek tudása szükséges a feladat megoldásához. A kép vagy a minta átméretezése során a pixelek száma megváltozik, és ilyenkor kérdés, hogy az új pixelek értéke mi legyen.

Erre a problémára szokták alkalmazni az interpolációt, ami a kép nem ismert értékeit közelíti az ismert értékek felhasználásával, így a lyukakat ki tudjuk tölteni értékekkel pl. nagyítás esetén. Az interpolációnak három alapvető technikája van: a legközelebbi szomszéd elve, a bilineáris és a bicubic interpoláció.

A feladat megoldásához az éldetektálást, a színszűrést, a mintakeresést, a kicsinyítés/nagyítás megoldásait fogom alkalmazni.

A megvalósítás terve és kivitelezése

A KRESZ táblák jellegzetessége, hogy megadott alakzatot (négyzet, nyolcszög, kör, téglalap) és megadott színt, színeket (piros, kék, sárga, fehér, fekete) színeket vesznek fel, illetve a mintájuk is adott.

Ezekből a tulajdonságokból kiindulva a programot úgy terveztem, hogy először a tábla színe szerint készítek szűrést a képre, amely után már a bináris képen csak az adott színű objektumok esetén keresem az éleket. Az élek megtalálása után a sokszög szögeinek számát vizsgálom, és ha az egyezik az általam keresett forma szögeinek számával (például elsőbbségadás kötelező tábla esetén hárommal), akkor egy mintakeresést is lefutttatok a képen. Ha a keresett minta és sokszög egymást nagy mértékben fedi, akkor elfogadom, hogy a keresett tábla ott található a képen, kiiratom a terminálra, hogy milyen táblát talált a program, és bekeretezéssel megmutatom a kimeneti képen, hogy hol találta az adott táblát, és azt milyen színnel keretezte be.

A kivitelezést úgy kezdtem, hogy elkezdtem Youtube-on nézni formakereséses videókat, hiszen mind az OpenCV, mind a Python új volt számomra, így a szükséges beépített függvények megismerését így láttam a legegyszerűbbnek. A programot 1-2 képpel folyamatosan teszteltem a készítés közben, hogy biztos az történik-e, amit szeretnék. Továbbá a kiiratások is több mindenre kiterjedtek.

A kód elején a szükséges csomagok behívását teszem meg, majd beolvasom a bemeneti képet és a keresett mintákat.

Amire nagyon hamar rájöttem, hogy a bemeneti kép mérete meghatározó, hiszen nem mindegy, hogy mekkora képen keresek egy bizonyos pixelszámú objektumot, ezért nagyon hamar bekerült a program elejére egy képátméretezés. A vizsgált kép mérete gyakorlatilag bármilyen nagy lehet, ezért egy while ciklust alkalmaztam, hogy addig felezze a szélességét és a magasságát a bemeneti képnek, amíg valamelyik a kettő közül nem lesz kisebb, mint 1800 képpont.

Ezután következik az a lépés, hogy az RGB képet átalakítom HSV képpé, amely alapja lesz a színek szűrésének. A színszűrésnél használt értékeket egy HSV térkép alapján állítottam be először, majd a vizsgált képek alapján finomhangoltam. Először a piros színre szerettem volna szűrni, de a HSV térképen láttam, hogy nem elég, ha a hue fenti értékeit veszem (150-180), ami a rózsaszín irányából megy a piros felé, hanem a lenti értékeket is be kell venni a maszkba (0-10), ami a pirosnak a narancssárgába hajló színeit fedi le. Ekkora skálára azért van szükség, mert a fényviszonyoknak, és tábla valós színének függvényében elég változatos lehet a képen a táblát megjelenítő terület színkódja. Így viszont két maszkot kellett készítenem, egyet a fenti értékekre, egyet pedig a lentiekre, viszont a piros maszkot együtt kell kezelje a program. Mivel mindkét maszk egy-egy bináris kép, ami ráadásul közös helyen nem is veszi fel a 255-ös értéket (hiszen a két skála között nincs átfedés), így a két maszk összeadása nem jelent problémát. Az összeg adja ki a végleges piros maszkot.

A kék és a sárga maszk elkészítése már nem volt ennyire bonyolult, hiszen ott nem kellett két részletben felírni a skálát.

A fehér maszk érdekessége, hogy itt a hue értéke a teljes skálát lefedi (0-180), és a saturation résznél van valójában leszűkítve “fehérre” a színkód (0-45).

A maszkok elkészítése után néhány változónak kezdőértéket adok. A táblák számát nullára állítom, így ha egy táblát sem talál a program, akkor ezt ki tudja írni. Az x és y a talált sokszög kezdőpontját, míg a w és a h a szélességét és magasságát jelöli. Ezeket a kezdőértékeket a későbbiekben felülírom a talált sokszög elhelyezkedése és mérete alapján. Továbbá egy logikai változót is bevezetek zsak néven, melynek kezdőértékét false-ra állítom. Ez a változó a piros színre való keresésnél válhat true-vá, abban az esetben, ha talál a program egy piros négyszöget a képet. A célom ezzel az, hogy megvizsgáljam, lehetséges-e, hogy van zsákutca tábla a képen. Amennyiben ez a változó false marad a kék szín keresésénél, abban az esetben nem fog zsákutcát találni a program.

Miután a kezdőértékeket megadtam, a piros színszűrést készítem el. Az egész képen végigiterálok, és azokat a területeket kijelöltetem a programmal, amik legalább 500 képpont területűek, és az általam meghatározott színt tartalmazzák (H: 0-10, 150-180 S: 100-255 V: 20-255). Mivel a fotózott képeken a sokszögek nem “szabályosak”, például egy elsőbbségadás tábla esetén a táblát rögzítő csavarok miatt nem teljesen háromszög alakú a piros rész, ezért készítem el az approx változót, ami a kontúrokat egyenletesebbé teszi. Ezekkel a “hozzávetőleges” formákkal tudok tovább menni, és megszámolni a talált alakzatok kontúrjai alapján az alakzatok éleinek számát. Amennyiben az alakzat éleinek száma három vagy nyolc, abban az esetben ezt a formát “bekeretezem” (ez a keretezés a kimeneti képen nem látszik), és a keret elhelyezkedésének és méretének megfelelően felülírom az x, y, w, és h értékeit.

Az eredeti terveim szerint ezután a háromszög esetében az elsőbbségadás kötelező táblát ábrázoló mintát, nyolcszög esetében a stop táblát ábrázoló mintát átméreteztem akkora méretűre, hogy az nagyjából megfeleljen a talált háromszög és nyolcszög méretének. Majd az átméretezett mintát a matchTemplate mintakereséssel megkerestem a teljes képen, és ha az elég közel esett a bekeretezett alakzathoz a teljes képen, akkor elfogadom, hogy van olyan tábla a képen, és ezt mind a terminálon megjelenítem, mind pedig a kimeneti képen jelölöm.

A tesztelések során azonban nagyon sokszor futottam bele abba a hibába, hogy a matchTemplate teljesen más területen mutatta a legnagyobb egyezőséget, és túl sok képen így nem működött megfelelően a program. Ezért ennél a két táblánál mást kellett alkalmazzak.

Az elsőbbség adás kötelező tábla esetén a piros háromszögön belül egy fehér háromszög is megtalálható, így itt a következő lépés a képen a fehér háromszög keresése a piros háromszög kereséséhez hasonlóan. Valójában ennél a fázisnál döntöttem úgy, hogy a piros, kék és sárga maszkokon kívül fehér szűrésre is készítek maszkot. Amennyiben a fehér háromszögnek kisebb a szélessége vagy a magassága, abban az esetben elfogadom, hogy elsőbbségadás tábla van a képen, és ezt a terminálon és a kimeneti képen is kiiratom.

A stop tábla esetében nem volt ilyen egyszerű alakzat, amire kereshettem volna és a matchTemplate sem működött, így itt a nyolcszög területét vizsgáltam tovább. Ezáltal például egy őszi képen a vörös leveleket kiszűri a program, nem jelöli be azokat stop táblának, akkor sem, ha nyolcszög alakzatot vesznek fel, viszont a tábla nem lehet túl kicsi (nem lehet túl

messze a fényképezőtől). Tehát ha a nyolcszög mérete megfelelően nagy, akkor elfogadom, hogy stop tábla van a képen, és ezt terminálon és a kimeneti képen is jelzem,

A piros szűrésnél még egy dolgot vizsgálok: ha találok minimum 200 képpont méretű négyszöget a képen, akkor a zsak változót true-ra állítom, vagyis lehetséges, hogy zsákutca tábla van a képen.

Ezután a kék szűrést végeztem el hasonlóan, mint a piros szűrést. A kék színt a következő értékek mentén kerestem: H: 95-130, S: 100-255, V: 20-255.

A kék szűrésnél szintén csak az 500 pixel méretűnél nagyobb területű alakzatokat vizsgáltam. Az alakzatok korábban leírt módon elkészített “hozzávetőleges” formájával továbblépve, a sokszög szögeit megszámoltam. Amennyiben kék négyszöget találtam, és a korábbi zsak változó true értéket vesz fel, abban az esetben a korábban a piros színnél is tervezett mintakereséssel mentem tovább. A kék négyszög méretének megfelelően átméreteztem a zsákutca minta képét, és azt a teljes képen kerestem a matchTemplate-tel. Ennél a mintánál nem volt jellemző a tesztelés során a fals eredmény, a keresett mintát jó helyre illesztette a program, így ezt a részét megtartottam a programnak a tesztelés után is. Amennyiben a minta találati helye és a négyszög helye némi hibahatárral egybeesik, akkor zsákutca tábla van a képen, és a terminálon és a kimeneti képen is jelölöm ezeket.

A kör esetében az approx értéke 8-at vesz fel, ezért ezen érték esetén vizsgálom tovább a képet a matchTemplate-tel a körforgalom mintáját alapul véve. Itt is hasonlóképpen a minta átméretezésre kerül a talált kör méretének megfelelően. A mintát megkeressük a teljes képen, és amennyiben ennek a kettőnek az elhelyezkedése megegyezik valamennyi hibahatárral, akkor elfogadom, hogy körforgalom tábla van a képen, és ezt a szokásos módon, a terminálon és a kimeneti képen is megmutatom.

Az utolsó színszűrés a sárga színre történik hasonlóképpen, mint az előző két szín esetében (H: 17-32, S: 100-255, V: 20-255). Amennyiben a sárga sokszög területe 500 pixelnél nagyobb, és négy szöge van, akkor a szokásos módon átméretezett minta teljes képen való keresése következik. A mintakeresés és a négyszög elhelyezkedése, egymástól való távolsága alapján dől el (akárcsak a kék táblák esetében), hogy az adott főútvonal táblát megtaláltnak tekintem-e. Amennyiben annak tekintem, abban az esetben ezt a terminálon és a kimeneti képen is jelzem.

A kód leírásának elején jeleztem, hogy van egy tablakszama változó a programben. Minden egyes tábla megtalálás esetén ennek a változónak a számát növelem eggyel. Amennyiben a program egyetlen táblát sem talál a képen, abban az esetben az kerül kiiratásra, hogy a program nem talált egyetlen táblát sem a keresett táblák közül.

A kód vége a kimeneti kép megmutatása, melyre a találatok közben bekeretezésre kerülnek a megtalált táblák. A kimeneti képen a kontúrok is megrajzolásra kerülnek vékony fekete színnel, hogy az ne vonja el a felhasználó figyelmét.

A program bezárása egy bármilyen gomb megnyomásával történik, ekkor zárja be a kimeneti képet is.

A program megírása közben sokszor ütköztem hibákba, éppen ezért a kód korábbi verziójában több kikommentelt rész is található, mely a különböző élhosszúságok terminálra való kiiratására, a maszkok megmutatására stb. szolgáltak. Mindezek segítettek a debugolásban, a hibák megtalálásában és javításában.

Tesztelés

A tesztelésnek két formáját is megvalósítottam. Az egyik az 1-2 képen való tesztelés magának a program megírása közben. Ennek a célja főként az volt, hogy a program tényleg azt csinálja-e, amit szeretnék. Ezt a fajta tesztelést a kód megírása közben, kb. 4-5 új sor bekerülése után futtattam. Már ezzel a módszerrel is nagyon sok hibát sikerült kiszűrnöm.

A fő tesztelés azonban akkor történt meg, amikor mennyiségi, 70 db képen teszteltem a programot. Sajnos a program nagyon sok képnél fals adatot mutatott, így vissza kellett térjek a program átírásához. Ekkor derült ki, hogy az elsőbbségadás táblánál és a stop táblánál a matchTemplate nagyon nagy hibaszázalék aránnyal dolgozik, így kénytelen voltam azt a részét a programnak teljesen átírni.

Ugyanitt derült ki, hogy a gyalogos táblát rendszeresen zsákutca táblának látja a program, így ez miatt került be a piros téglalap (négyszög) keresése is a programba. Ezzel szerencsére sikerült megszüntetnem, redukálnom ezeket a hibákat.

A színek finomhangolását is a mennyiségi adat alapján tudtam elkészíteni, és a program végleges verziójában sem lett ez tökéletes. Az időjárás viszontagságainak kitett táblák elkezdenek fakulni, és a színűk már jobban közelít a fehérhez, mint az eredeti színűkhöz. Ha

viszont az ilyen színekre is kiterjesztem a keresést, akkor a maszk túl sok olyan képpontra kerül rá, aminél egyáltalán nincs tábla, és azt sem szerettem volna, hogy ez miatt olyan helyeken mutasson táblát a program, ahol egyáltalán nincs semmilyen tábla.

A képek keresésénél, elkészítésénél próbáltam változatos forrásokat, képeket használni. Vannak olyan képek, amik eleve kisebb méretűek, vannak nagyobb képek, ez utóbbiak általában saját készítésűek. Amikor ezeket teszteltem, akkor jöttem rá, hogy a bemeneti képet is át kell méretezzem, hogy a program jobban működjön. A tesztképeknél a fényviszonyok változatosak, vannak napos időben, ködös időben, este készült képek is.

Ezen felül kerestem és készítettem "becsapós" képeket is, mert kíváncsi voltam, hogy ezekre hogyan reagál a program. Így készült olyan kép is, amelyiken nincs olyan tábla, amire a program kiterjed, illetve olyan is, ahol a keresett táblákhoz nagyon hasonló objektum található a képen. Ilyen például egy kéz, melyre egy stop tábla van festve, vagy egy tanuló autó, ahol a T betű miatt a zsákutcás tábla látható benne. Sajnos a becsapós képek azon fajtái, amelyeknél egy táblára hasonlító objektum van a képen, megvezetik a programomat.

Összességében azt gondoltam, hogy a tesztelés folyamata egyszerűbb lesz. Számítottam néhány hibára, amelyet a mennyiségi képek előhoznak, de nem gondoltam volna, hogy a programot helyenként alapjaiban kell megváltoztatnom. A program így most körülbelül 70%-osan megbízható, ami az ipari megvalósításoktól nagyon messze áll, de azt gondolom, hogy a programom jó alapja lehet egy valós környezetben is használt közlekedési tábla felismerő programnak, természetesen kiegészítve, jobban finomhangolva, ami úgy már nagyobb megbízhatósággal bír.

Felhasználói leírás

A programot Fedora operációs rendszeren készítettem, így azt szeretném bemutatni ebben a fejezetben, hogy egy hasonlóan Linux alapú operációs rendszeren hogyan kell használni a szoftvert. A Linuxokon a Python programot nem kell külön feltelepíteni, viszont néhány egyéb csomagot telepíteni kell, hogy működjön a program. A szükséges csomagok:

- python3-opencv.x86_64,
- python3-numpy.

Ezeket terminálon a yum install (Fedora/Red Hat) / apt install (Ubuntu) / zypper install (SUSE) parancs után beillesztve a csomag nevét meg lehet tenni. Az install parancs pontos kifejezése attól függ, hogy milyen Linux alapú operációs rendszert használunk. Amennyiben további csomagok szükségesek ezek “dependency” csomagjaiként, akkor azt a terminálon jelezni fogja az OS, és azok telepítését is el kell fogadni.

Ezután magába a programba kell beleírni mind a vizsgált kép, mind a keresett minták teljes elérési útvonalát, ezek a path-ok mindegyik alábbi sorban a (“ karakterek után és a “) karakterek elé kell, hogy kerüljön:

- 8. sorban kell beírni a keresett kép útvonalát,
- 10. sorban a zsákutca minta kép útvonalát kell beilleszteni,
- 11. sorban a körforgalom minta kép útvonalát kell beilleszteni,
- 12. sorban pedig a főútvonal tábla minta kép útvonalát kell beilleszteni.

Amennyiben ez megtörtént, a program készen áll a futtatásra. A futtatást úgy lehet megtenni, hogy a terminálon beírjuk, hogy python szót, majd a program teljes elérési útvonalát. Amennyiben abban a mappában állunk, ahol a programfájl található, akkor elég annak a nevét beírni a python szó után. Az Entert leütve a program lefut és kiírja a terminálra a talált tábla nevét, és hogy milyen színnel van bekeretezve a kimeneti képen, vagy pedig azt, hogy nem talált a szoftver a keresett táblák közül egyet sem.

A program bezárásánál fontos, hogy a kimeneti képet ne zárjuk be, hanem a billentyűzeten egy gombot nyomjunk meg, amikor a kimeneti kép az aktív ablak. Ezzel lehet a program futását befejezni, így a program kilép és bezárja a kimeneti képet.

Irodalomjegyzék

- A tárgyhoz tartozó elméleti diasorok
- https://github.com/horverno/sze-academic-python/commit/07e96fe1232aac499cd674e9dea527387b66ff5c
- https://cvexplained.wordpress.com/2020/04/28/color-detection-hsv/
- https://www.youtube.com/watch?v=43pCXboZ5hE
- https://www.youtube.com/watch?v=Fchzk1lDt7Q