Gépi látás

Diagram felismerés

Józsa Dávid

R629Q7

Győr, 2020/21

Tartalomjegyzék

1 Bevezetés

Szoftverfejlesztési, gyártási folyamatokat, valamint cégek felépítését is lehet ábrázolni diagrammokon, amelyek átláthatóságot biztosítanak az összetetten működő folyamatokról. Segíthet egy folyamat tervezési, kivitelezési fázisban, valamint a dokumentációban, az egyes részeinek megértésében. Ez a beadandó feladat bemutat egy olyan szoftvert, amely képes felismerni a bemenetére adott képről egy diagramot, majd kimenetként egy file-t hoz létre, amely által a diagram tovább szerkeszthető.

A dokumentáció először bemutatja a szoftver, algoritmusok elméleti hátterét, bemutatva az alkalmazott eszközöket, a program bemenetét, kimenetét, valamint előre mutat az alkalmazott megoldások előnyeire, hátrányaira. Ezután bemutatásra kerül a kivitelezés, a program pontos működése, részletesen bemutatva az alkalmazott algoritmusokat. A végén, a tesztelés a program működését teszi próbára, hogy különböző bemeneteknél milyen kimenetet produkál. Ebből értékes információt lehet kinyerni a működés hatékonyságáról, pontosságáról, valamint megbízhatóságáról.

2 Elméleti háttér

2.1 Felhasznált eszközök

A szoftver **python** programozási nyelven készült. A képfelismerési feladatokra az **opencv (ver. 4.4.0.44) [9]** szoftvercsomag került felhasználásra, amely a **numpy (ver. 1.19.2) [10]** csomaggal kombinálva különféle tárolási megoldások, és függvények biztosításával segítette a projekt megvalósítását. Ezen kívül használt csomagok még a **sys**, az argumentumok olvasásához, valamint az **os.path**, amely segítségével a bemeneti fájlt lehet ellenőrizni.

2.2 Diagram felismerése

2.2.1 Felismerés folyamata

A diagramok segítségével információkat jeleníthetünk meg ábrán szemléltetve. A program célja különféle, szöveg nélküli, folyamatábra stílusú diagramokat felismerni, előre meghatározott komponenseket keresve a bemeneti képen. Ezen komponensek 2 nagy csoportba sorolhatóak:

- alakzatok
- nyilak

A cél a bemeneti képen található diagramról felismerni a rajta lévő alakzatok típusát, méretét és pozícióját, valamint az ezen alakzatokat összekötő nyilakat. Itt fontos megemlíteni, hogy a program megfelelő működéséhez a bemenetnek meg kell felelni különféle megkötéseknek, melyek később kerülnek részletezésre.

2.2.2 A program által felismerhető diagrammok

A programnak van néhány megkötése a felismeréssel kapcsolatban. A legfontosabb megkötés, hogy a szoftver előre meghatározott alakzatokat tud felismerni a képen. Ezen alakzatok a következők:

- Háromszög
- Téglalap
- Rombusz
- Paralelogramma
- Ellipszis

Ezen kívül a nyilakra is van két megkötés:

- A nyilak csak függőleges és vízszintes vonalakból állhatnak
- A nyilak hegyének tömör háromszög szerű alakzatnak kell lennie a megfelelő felismerhetőség érdekében

További megkötés, hogy a vonalvastagsága a diagram elemeinek mindenütt ugyanakkora legyen, valamint, hogy a nem érintkező vonalszakaszok vége legalább 30 pixellel eltávolodva legyenek egymással. Ezen megkötések nélkül a program a diagrammokat nem megfelelően ismerheti fel.

2.3 A szoftverben használt képfelismerési megoldások

2.3.1 Általános bemutatás

A szoftverben használt megoldások pixelműveletekre hagyatkoznak. Ez azt jelenti, hogy a programban az algoritmusok a képen található pixelek információira hagyatkoznak, vagyis a pixelek értékére, és pozíciójára. Ezen információk a programban különféle módon feldolgozásra kerülnek, legyen az egyedileg megírt algoritmus, vagy az opencv által biztosított függvény.

2.3.2 Előfeldolgozás

Az algoritmusok működéséhez szükséges, hogy a bementei képet különféle módszerekkel átalakítsuk, hogy könnyebben feldolgozható legyen. Ez a folyamat az előfeldolgozás (preprocessing).

Az előfeldolgozás folyamatába tartozik a kép szürkeárnyalatossá, majd binárissá, azaz fekete-fehérré alakítása (binarization / thresholding), a kép felismerhetőségének növelése, valamint az esetleges hibák javítása, mint például a nem teljesen összeérő vonalak javítása.

2.3.3 Összefüggő komponensek

Az összefüggő komponensek (connected components) algoritmus segítségével meghatározhatóak olyan területek egy képen, amely pixelei ugyanolyan értékkel rendelkeznek, ezáltal hasznosnak bizonyulva a diagramot alkotó alakzatok megkereséséhez.

A függvény működésekor pixeleket, és azok szomszédjait vizsgálja. Egy pixel és egy szomszédja akkor összefüggő, ha értékük megegyezik. Vizsgálat közben az algoritmus kétféle módszerrel dolgozhat, a 4 és a 8 irányú kapcsolódási vizsgálattal. Ez azt határozza meg, hogy a vizsgált pixel szomszédjai közül négyet vagy nyolcat vizsgáljon. Ennek tudatában végig lehet vizsgálni a képet, meghatározva az egybefüggő területeket. [2]

2.3.4 Kontúr detektálás

A programban sok helyen hasznosítva volt a kontúr detektálás. A kontúr egy olyan görbe, amely kisebb szakaszokból épül fel, és azok ugyanolyan értékű pontokból állnak. Segítségével meghatározható egy objektumot körbevevő határolóvonal (körvonal).

Kontúrok többféleképpen is leírhatóak, amelyből a leggyakoribb módszer a Freeman lánckód, vagy annak valamilyen átalakítása. A módszer esetén a kontúr pontjai egymás után, láncba sorolva vannak megadva, ezáltal felépítve a körvonalat. [1]

2.3.5 Kimeneti file

A program a felismerés után kirajzolja a képernyőre az eredeti képet, azon kiemelve a felismert komponenseket, valamint létrehoz egy filet. A file xml formátumú grafikai modell, amely a felismert alakzatokat, nyilakat cellákban tartalmazza. Ez a file az app.diagrams.net oldalon beimportálható, valamint tovább szerkeszthető.

3 Tervezés és kivitelezés

3.1 A program felépítése

3.1.1 Beolvasás és előfeldolgozás

A program a bemenetét, amely egy képfile elérési útvonala, parancssori argumentumként kéri be. Miután elindul a program, megvizsgálja, hogy meg lett-e adva pontosan 1 argumentum, és hogy az a file létezik-e. Ha átment az ellenőrzésen, akkor a kép beolvasásra kerül az opencv *imread* függvényével. A kép, és a programban az összes további kép is egy numpy tömbben tárolódik. A beolvasott képet utána szürkeárnyalatossá alakítja a *cvtColor* függvény segítségével, majd legvégül ezt binarizálja, ezáltal minden pixel csak két értéket vehet fel.

A binarizálás a *threshold* függvény segítségével zajlik le. A *threshold* függvény paraméterezésekor meg kell adni a szürkeárnyalatos bemeneti képet, 2 értéket, amelyek segítségével kiválogatja a pixeleket, valamint a módszer, amivel válogat. A megadott 2 számérték közül az első adja meg a threshold értéket (határérték). Ez az érték lesz az elválasztó érték, amely segítségével értékelhető a pixel az alapján, hogy a pixel értéke a határérték alatt vagy felett található. A második érték a maximum érték. Ez az érték lesz hozzárendelve azokhoz a pixelekhez, amelyek túllépik a határértéket. A programban ez a két érték a 150 és a 255, ami azt jelenti, hogy minden, 150 feletti értékhez 255 értéket fog rendelni. [4]

A binarizált képen utána még végre van hajtva két morfológiai transzformáció, méghozzá a morfológiai nyitás és az erózió. A nyitás által az esetleges vonalak hibái, kimaradásai vannak javítva, az erosion következtében pedig a vonalak vastagodnak meg a könnyebb felismerés érdekében. Ez alapvetően rossznak hangzik, hiszen az erózió a vonalak szűkítésére használatos, a szakaszok javítására pedig a morfológiai zárás használandó. Ez azonban csak akkor igaz, ha a képen a háttér fekete, és az objektumok fehérek, azonban jelen esetben ez fordítva igaz, ezért működik az alkalmazott megoldás. [5]

3.1.2 Alakzatok felismerése

Az előfeldolgozás után az alakzatok felismerése következik. A felismerés itt az összefüggő komponensek módszerre hagyatkozik. Ez a funkció az opencv *connectedComponents* függvényével elérhető, amelynek csak egy bemeneti képet kell megadni, és visszaadja a talált komponensek címkéit (komponensek száma), és egy címkézett képet. A címkézett kép egy, az eredeti képpel megegyező méretű mátrix, amelyben a megtalált komponensek fel vannak címkézve számokkal. Ez azt jelenti, hogy a képen egy adott komponens összes pixelének értékét az adott címke értékére állítja. [6]

Miután felcímkézte a képet, azokon egyesével végighalad a program. Minden komponenshez létrehoz egy maszkot, amely háttere fekete színű, és az adott komponens pixeljei fehérek. Utána ezen a képen végrehajtható egy kontúr keresés, amely megadja a maszkban található fehér rész, azaz a talált komponens körvonalát.

A kontúron végrehajtódik az *approxPolyDP* függvény, amely a Douglas-Peucker algoritmus segítségével közelíti a sokszöget, vagyis leegyszerűsíti a görbéket. Ennek az egyszerűsítésnek a pontossága megadható egy szám formájában, amely minél kisebb, annál érzékenyebb a kisebb változásokra a görbében, vagyis annál kevésbé egyszerűsít. Ezután megállapításra kerül a komponens helye, és mérete. Ez a *boundingRect* függvény segítségével történik, amely meghatározza a kontúr köré írható téglalap pozícióját és méretét. [6]

Ezután meg van vizsgálva, hogy a meghatározott, közelített kontúrok által leírt alakzatnak hány oldala van. Az oldalak száma alapján beazonosítható, hogy az alakzat háromszög, négyszög, vagy ellipszis.

A négyszögeknél a háromféle különböző négyszöget a kontúr területének és az azt körbevevő téglalap területének arányából következteti ki a következőképp:

- Ha az arányuk maximum 5%-kal tér el, akkor téglalap
- Ha a kontúr a fele a körbevevő téglalaphoz képest (5% hibahatárral), akkor rombusz
- Különben paralelogramma

Felismerés után egy tömbbe le van tárolva a terület pozíciója, mérete, egy karakter által megjelölve a típusa, valamint az alakzat kontúrja.

Az alakzat felismerése után még egy dolga van a programnak. Mivel olyan terület is összefüggő komponensnek számít, amelyet a diagramm esetében nyilak, és alakzatok kerítenek be, ezeket az alakzatokat ki kell szűrni. Emiatt az alakzat felismerése után a program átnézi, hogy a felismert alakzat átfedésben van-e az eddig eltárol alakzatok egyikével. Ha talál átfedést, akkor kitörli azt a komponenst, amelyik területe nagyobb, tehát azt, amelyik tartalmazza a másik alakzatot. Ennél az algoritmusnál nem kell a méreten kívül mást ellenőrizni, mivel egy diagrammban az alakzatok nem fedhetik egymást, ezért ilyen eset csak akkor fordulhat elő, ha egy komponenst tévesen alakzatnak ismertünk fel.

3.1.3 Nyilak felismerése

3.1.3.1 Nyilak és alakzatok elkülönítése

A nyilak felismerésének folyamata egy új kép létrehozásával kezdődik. Ez a kép az eredeti kép binarizált változatából indul ki, amelyen kitöltésre kerülnek az eddig már megtalált alakzatok. Mivel az alakzatok egy olyan képen lettek keresve, amelyen az élek meg lettek vastagítva, ezért a talált alakzatok mérete kisebb, mint az eredeti képen. Emiatt a megtalált kontúrok kitöltésén kívül végre kell hajtani egy morfológiai zárást, hogy eltűntessük az eredeti diagram körvonala, és a kitöltött kontúrok közötti közt. A zárás megfelelő működéséhez a kép előtte invertálásra kerül, azaz a háttér fehér, a diagram elemek fehérek lesznek.

Ezután következik az egyik legfőbb lépés a nyilak felismeréséhez, az alakzatok, és a nyilak szétválasztása. Ennek megvalósításához ismét morfológiai transzformációk lettek felhasználva. A folyamat úgy zajlik le, hogy az imént létrehozott képen, amely az egész, kitöltött diagrammot tartalmazza, addig hajtunk végre eróziót, ameddig nem tűnik el minden, az alakzatokat kivéve. Ennek ellenőrzése a képen található kontúrok számának vizsgálatával történik. A folyamat közben megszámlálásra kerül az, hogy hány iteráció kellett a nyilak eltűntetéséhez. (A megoldást inspirálta: [7])

Az erode iterációk megszámlálása után létrehozható a csak nyilakból álló kép. Először létrehoz egy maszkot, amely az alakzatokat fedi le. Ez úgy hajtja végre, hogy a képre, amelyen az erózió végre lett hajtva, kirajzolja a kontúrokat úgy, hogy kitöltse az alakzatokat, valamint a szélét a megszámolt iterációval arányosan vastagítja meg. Erre azért van szükség, mert az alakzatok kontúrjai nem tartalmazzák az alakzatok eredeti körvonalát, csak az alakzatok belső részét.

3.1.3.2 Nyilak pozíciójának, méretének meghatározása

A nyilak beazonosítása az elszeparált nyílkép szkeletonizációjával (skeletonization) kezdődik. A szkeletonizálás által a nyilak vastagságától függetlenül egy olyan képet kapunk, amelyen a nyilakat 1 pixel vastag vonalak képviselik. Ebből a képből könnyen kivehetőek a vízszintes és függőleges vonalkomponensek azzal, hogy a képben átfuttatunk két kernelt. Az egyik

vízszintesen, a másik függőlegesen tartalmaz 1-es értékeket egy sorban/oszlopban, valamint az egyesek által alkotott vonallal párhuzamos szomszédokban 0-kat. A kernelekkel át lehet szűrni a szkeletonizált képet, amely által két olyan képet kapunk, amelyen csak a nyilaknak a vonal részei találhatóak, kiszűrve a zajokat, az alakzatokból visszamaradt pixeleket, valamint a nyílhegyeket. Ezeken a képen kontúrkereséssel megtalálhatóak az egyes vonalszakaszok, azok pozíciója, mérete, amelyek tárolásra kerülnek.

A vonalak megkeresése után meghatározásra kerülnek a vonalak végpozíciói, valamint azok a vonalak, amelyek keresztezik egymást. A program szempontjából azt a pontot értjük végpozíciónak, amely egy vonal vége, és egy adott határértékhez mérten közel van egy alakzathoz. Ennek meghatározásához az algoritmus végigmegy minden vonal végén, és ellenőrzi, hogy egy területnek a határértékkel megnagyobbított dobozában (hitbox) van-e a végződés. Ha igen, akkor elmenti végpontként, valamint eltárolja, hogy melyik vonal, melyik alakzathoz csatlakozik, valamint, hogy a vonal milyen irányból közelíti meg az alakzatot, aminek a kimeneti file létrehozásánál lesz szerepe.

A vonal kereszteződések meghatározása hasonlóan történik, mint a végpontok megkeresése, azzal a különbséggel, hogy ezesetben a vonalak végpontjainak pozícióját nem alakzatok, hanem más vonalak végéhez hasonlítjuk. Ha találunk olyan pontot, amely közelében több mint 2 vonalvég található, akkor azt a pontot elmentjük kereszteződésként. Ezen módszer miatt van a megkötés, hogy 30 pixel legyen a nem érintkező vonalszakaszok végpontjai között, mivel a közelségvizsgálatnak van egy határértéke, amely 30 pixelre van állítva. A határérték azt a távolságot adja meg, hogy egy végpont egy másik végponttól maximum milyen távol lehet, hogy azokat még egy pozícióban lévőnek lássa.

3.1.3.3 Nyilak által összekötött alakzatok meghatározása

Az összeköttetések meghatározása kétfelé van választva: először azokat a vonalakat vizsgálja a program, amelyek nem tagjai kereszteződéseknek, utána azokat, amelyek tagjai.

Az egyszerű vonalak esetében lefut egy algoritmus, amely az egyik végpontból indul ki. Megkeresi ennek a végpontnak a másik felét (a vonal szakaszának másik oldalát), majd elkezd keresni hozzá közel álló vonalvégződést. Ha talál olyat, amelyik nem önmaga, akkor annak is megkeresi a másik felét, és halad tovább. Ezt a folyamatot addig ismétli, amíg el nem jut egy másik végpontig, tehát amíg nem találja meg a másik alakzatot. Amint megtalálta, elmenti a kezdő és a végpozíciót, valamint az összekötött alakzatok indexét.

A kereszteződésen áthaladó vonalak meghatározása hasonló elven működik, azonban itt több mint 2 végpontot kell összekötni. Az algoritmus ezesetben a kereszteződés pontjából indul ki, és az előző megoldáshoz hasonló módszerrel halad végig a vonalszakaszokon. A folyamatot azonban egy rekurzív függvény valósítja meg. Erre abban az esetben van szükség, ha nem csak egy, hanem két vagy több kereszteződés is van a diagramban, és az a kettő összeköttetésben van egymással.

3.1.3.4 Nyílhegyek keresése

Az utolsó feladat a nyilak meghatározásánál a nyilak hegyeinek meghatározása, tehát hogy az egyes vonalak melyik végén található. Ehhez a program létrehoz egy képet, amely (nagyrészt) csak a nyílhegyeket tartalmazza. Ezt úgy érheti el, hogy az eredeti képből kivonja a korábban készült, csak az alakzatokat tartalmazó képet, valamint erre feketével átfesti a vonalszakaszokat, ezáltal csak a vonalhegyek maradnak. Ilyenkor nem maradt más, mint megnézni, hogy a teljes vonalak egyik és másik fele körül mennyi pixel van ezen a képen. Amelyik oldalán több pixel van, az lesz a nyílhegy.

Kereszteződéses vonalak esetén is hasonló a módszer. Ilyen esetben mindig, a kereszteződés egyik végpontja tér el a többitől, tehát van egy forrásból vezet nyíl sokfelé, vagy sok forrás mutat egy helyre. Ilyen esetben úgy dönti el, hogy melyik forrás és melyik nem, hogy minden végződés pixelszámainak átlagát veszi alapul. Ha egy végpont átlag alatti pixelt tartalmaz, akkor forrás, ha átlag felettit, akkor cél.

3.1.4 A program kimenete

A programnak kétféle kimenete van. Az első egy képernyőre rajzolás, amely esetén egy grafikus ablakban megjelenik a beolvasott diagram, valamint a különböző komponensei rárajzolva.

A második kimenete egy xml file, amely a javascriptben megvalósítható mxGraph modell szintaxisát használja. Ezt a szintaxist az *app.diagrams.net* weboldal fel tudja dolgozni, és lehetőséget biztosít a diagram szerkesztésére.

Az modellben a diagram elemei külön rétegekben (layer) vannak tárolva. Minden rétegben megadható egy komponens típusa, valamint az ahhoz tartozó információk (pozíció, méret stb.). Ezen kívül minden komponenshez tartozik egy id. Ez az id egy szöveges azonosító az adott elemhez, amely segítségével beazonosíthatóak a komponensek. Ez a nyilak megadásánál játszik nagy szerepet, mivel a nyilak esetében a kezdő-, és végalakzat azonosítóját kell megadni, és a program magától kirajzolja a vonalat. [3]

4 Tesztelés

4.1 A tesztelésről általánosan

A program tesztelése manuálisan lett végrehajtva, mivel a kimenet túl összetett az automatizáláshoz. A tesztek során különféle véletlenszerű diagram, valamint többféle specifikusan alkotott diagramot teszteltem a programmal, amelyek célja, hogy a szélsőséges eseteknél is próbára legyen téve a program.

4.2 A tesztelt képek

A tesztek nagy része digitálisan készült diagram, valamint egy része kézzel rajzolt, és fotózott kép. A diagramok készítésekor figyeltem arra, hogy minél többféle szélsőséges esetet képviseljenek a tesztképek. A szélsőségek az alábbi variációkkal készültek:

- Vonalak vastagságának variálása
- Az alakzatok elnyújtása, deformálása
- A kép elforgatása

4.3 Teszteredmények

4.3.1 Program megbízhatósága

A tesztelés során a programban nem sikerült futás idejű hibát generálni. A tesztek 2%-a esetén nem volt érdemlegesen sikeres a felismerés, 22%-ban részlegesen sikeres, és 76%-ban sikeres volt a felismerés.

1. ábra: Tesztek eredménye

4.3.2 A tesztekből levont következtetések

A tesztelés során a program legtöbb funkciója, hibalehetősége meg lett vizsgálva. Az eredményekből elmondható, hogy a program első szakasza, azaz az alakzatok felismerése nagyon megbízhatóan működik. A tesztek majdnem egészében helyesen felismerte az alakzatokat, kivéve ahol nagyobb mértékben hiányos volt egy vagy több alakzat vonalszakaszai. Ilyen esetben azokat az alakaztokat nem ismerte fel a program, azonban a futást nem akadályozta, a többi alakzatot sikeresen felismerte. Minden más esetben nem volt gondja a programnak az alakzatokkal, felismerte azokat nyújtás, forgatás, valamint a vastagságuk variálása esetén is.

A tesztek alapján a nyilak felismerésével volt több probléma, azonban nagyrészt helyt állt a szoftver. A nyilak felimerésének teszteléséhez először különféle vastagsággal (1-5 pixel) készült diagramok lettek vizsgálva. A tesztek alapján a vonal vastagságát jól kezeli a program, nem volt probléma a nyilak felismerésével, még akkor sem, amikor egy diagramon belül változott 1-2 pixel intervallumban a vonalvastagság

Azoknál a teszteknél, amelyeknél a diagram el lett fogatva, a vonalfelismerés megbukott, mivel a program vízszintes és függőleges vonalszakaszokat keres a képen.

Az alakzatok nyújtása is hatással volt a nyilak felismerésére. Ilyen esetekben a vonalak megtalálásával nem volt probléma, viszont a nyílhegyek beazonosítása során néhol tévedett a program. Ez olyan esetekben fordult elő, ahol úgy voltak elnyújtva alakzatok, hogy ahol a nyilak csatlakoznak hozzájuk, ott az alakzatnak hegyesszög kiemelkedése van. Ennek a legegyszerűbb esete, amikor egy háromszög hegyéből indul ki egy vonal.

Ez azért tud probléma lenni, mivel a program a nyílhegy kereséskor a nyilak végét vizsgálja, hogy melyik körül található több pixel. Ez alapvetően nem lenne probléma, mivel ez a keresés elméletben egy olyan képen van végrehajtva, amely csak a nyílhegyeket tartalmazza. Erre a képre azonban kerülhetnek nem odaillő pixelek a hegyesszögekből, mivel az alakzatok kivágásakor a program úgy tűnteti el az alakzatok körvonalát, hogy adott vastagsággal kirajzolja a kontúrját a háttér színével. Ilyenkor azonban a szögeknél a kirajzolás eltompul, otthagyva a hegyesszög egy részét a képen, amelyet az algoritmus beszámol nyílhegynek. Ez a probléma általában úgy jelenik meg, hogy felismeri helyesen egy nyílnak a menetét, azonban fordított irányba nézve menti el.

Ezen kívül a kézzel rajzolt ábrák esetén is gyakran hibásan működik a nyílfelismerés. Ennek egyik oka, hogy a kézzel rajzolt ábrákon nehéz egyenes vonalakat rajzolni, valamint a kép készítésekor se biztos, hogy egyenes a kép. Ilyenkor a nyilak felismerése általában rosszul, vagy egyáltalán nem történik meg.

4.4 Fejlesztési lehetőségek

A tesztelés rámutatott néhány problémájára a programnak. Ebben a szekcióban röviden rámutatok javítási lehetőségekre, amelyek segítségével javítható a program megbízhatósága, hatékonysága.

A forgatási problémák kiküszöbölésére megoldás lehet, ha a beolvasott képet nem csak az alap orientációjában vizsgálnánk, hanem addig forgatva is, amíg nem találunk vonalszakaszokat. Ennek problémája viszont a nagy teljesítményigény, hiszen minél többet kell forgatni egy képet, annál többször kell átvizsgálni az egészet, vonalszakaszokat keresve.

A teszteléskor többször is probléma volt, hogy a nyílfelismeréskor a vonalaknak egyenes szakaszoknak kell lennie. Ez kiküszöbölhető lehet azzal, hogy ha nem vízszintes és függőleges kernelek szűrésével keresünk vonalakat, hanem a Hough transzformáció segítségével. A standard Hough transform segítségével a vonal orientáltságától függetlenül kereshetőek vonalszakaszok, továbbá a valószínűségi Hough transzformáció segítségével a kézzel rajzolt diagramok vonalszakaszai is nagyobb hatékonysággal felismerhetővé válnak. [8]

5 Felhasználói leírás

Github repository: *https://github.com/AnonymDavid/Diagram_szkenneles*

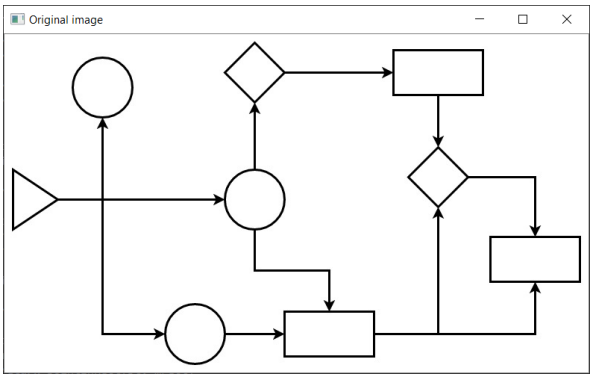
A program használatához a github repository-t kell leklónozni, valamint a korábban közölt python szoftvercsomagokat kell telepíteni. A programot egy file foglalja magában, ez a „**diagram.py**”.

A program indításakor egy paramétert fogad a program, amelyet parancssori argumentumként kell megadni. Ennek a paraméternek egy képnek kell lennie, és meg kell adni az abszolút elérési útját, vagy a program mappájától vett relatív elérési útját. Ezt a képet a program beolvassa, majd feldolgozás után parancssori üzenetben közli, amikor kész van egy „DONE” üzenettel, valamint megjelenít 2 ablakot. Az első ablak az eredeti beolvasott képet mutatja. A második ablak is a bemeneti képet mutatja, azonban erre a képre rá lettek rajzolva a felismert komponensek, különféle színkódolással:

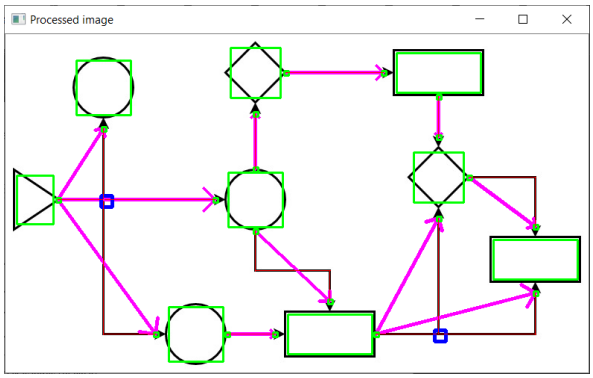
- Zöld téglalappal vannak bekeretezve a megtalált alakzatok
- Piros vékony vonallal vannak átfestve a vonalszakaszok
- Kék téglalappal vannak bekeretezve a nyíl kereszteződések
- Lila vastag vonallal vannak berajzolva az alakzatok közötti teljes vonalutak

A megjelenített képeken kívül a program létrehoz egy **output.xml** fájlt a programmal egy mappába. Ez a file beimportálható az ***app.diagrams.net*** oldalon, ahol a diagram tovább szerkeszthető az oldal eszközeivel.

A következő képeken egy példa látható a program ablakokban megjelenített kimenetére:



2. ábra: Bemeneti kép



3. ábra: Feldolgozott kép

6 Irodalomjegyzék

[1] Maurício Marengoni and Denise Stringhini (2011): High Level Computer Vision Using OpenCV. In: 24th SIBGRAPI Conference on Graphics, Patterns, and Images Tutorials, p. 11-24

[2] Prateek Joshi, David Millan Escriva and Vinicius Godoy (2016): OpenCV By Example.

[3] mxGraphModel documentation (Visited: 2020.12.24): *https://jgraph.github.io/mxgraph/docs/js-api/files/model/mxGraphModel-js.html*

[4] Thresholding (Visited: 2020.12.22): *https://docs.opencv.org/master/d7/d4d/tutorial_py_thresholding.html*

[5] Morphological transformations (Visited: 2020.12.22): *https://docs.opencv.org/master/d9/d61/tutorial_py_morphological_ops.html*

[6] Structural Analysis and Shape Descriptors (Visited: 2020.12.22): *https://docs.opencv.org/master/d3/dc0/group__imgproc__shape.html*

[7] Recognizing graphs from images (Visited: 2020.12.22): *https://www.yworks.com/blog/projects-optical-graph-recognition*

[8] Feature detection (Visited: 2020.12.22): *https://docs.opencv.org/master/dd/d1a/group__imgproc__feature.html*

[9] Opencv (Retrieved: 2020. oct.): https://pypi.org/project/opencv-contrib-python/

[10] Numpy (Retrieved: 2020. oct.): https://pypi.org/project/numpy/