Szegedi Tudományegyetem

Informatikai Intézet

SZAKDOLGOZAT

Szelepcsényi Dávid

2025

Szegedi Tudományegyetem

Informatikai Intézet

**Kézzel rajzolt E-K diagram digitalizálása**

Szakdolgozat

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Készítette: |  | Témavezető: |  |
|  | Szelepcsényi Dávid |  | Dr. Kardos Péter |  |
|  | Programtervező Informatikus BSc szakos hallgató |  | adjunktus |  |

Szeged

2025

## Feladatkiírás

A feladat egy olyan program készítése, ami egy raszteres képet kap, és azon végzett műveletekkel meghatározza a diagram elemeit, majd egy XML nyelvű, vektoros rajzolóprogram fájlformátumába átkonvertálni. Ez az OpenCV Python csomag segítségével érjük el. A program képes kell, hogy legyen felismerni az egyed kapcsolat diagram elemeit, azok közt a kapcsolatokat meghatározni. A program, a PyTesseract csomag segítségével képes a szövegek felismerésére. A program futása fél-automatikus, a felhasználó képes átírni a talált szöveget, illetve a felhasználó határozza meg, hogy melyik elem számít gyenge elemnek vagy gyenge kapcsolatnak.

## Tartalmi összefoglaló

* A téma megnevezése:

E-K diagram raszteres képének beolvasása, majd a diagramot leíró, vektorgrafikus diagram szerkesztő program által szerkeszthető fájl készítése.

* A megadott feladat megfogalmazása:

Egy olyan program készítése, amely képes E-K diagram elemeit, és a köztük lévő kapcsolatokat felismerni egy, a programnak megadott kép alapján, majd az így összegyűjtött információ alapján egy XML forráskódú fájlt állít elő

* A megoldási mód:

Python programozási nyelv használatával valósítottam meg a feladatot. Amennyiben az input kép megfelel az előfeltételeknek (pl. világos háttéren sötét szín objektumok vannak-e) előfeldolgozási műveletekre kerül sor. Után, az előkészített képen, néhány alakjellemző vizsgálata alapján megállapítom a rajzelemek típusát, méreteit, koordinátait. Az ebből nyert információk, illetve az eredeti kép alapján meghatározom az elemek között lévő kapcsolatokat. A jellemző, talált kapcsolatok, illetve a felhasználó által megadott paraméterek alapján előáll egy .drawio kiterjedésű fájlt.

* Alkalmazott eszközök, módszerek:

Az implementációhoz Python 3.12.3.-as verziót, és OpenCV, kép kezelésre kitalált Python könyvtárat használta. A vizuális felülethez PyQt6 grafikus csomagot, a szövegek felismerésére PyTesseract függvény könyvtáratat használta. Az XML fájlok a Python beépített, erre a célra szolgáló xml csomag segítségével készültek. A szükséges számítások elvégzéséhez a program NumPy függvény könyvtárat, és a beépített math csomagokat alkalmaz.

* Elért eredmények:

A program képes felismerni az E-K diagram különböző elemeit, és az azok között fennálló kapcsolatokat. A szövegfelismerés hibája esetén a felhasználó képes javítani a hibákat, illetve képes meghatározni gyenge elemeket, kapcsolatokat, valamint megadni a diagram kulcsait.

* Kulcsszavak:

E-K diagram, Python, OpenCV, XML, PyTesseract,

## Tartalomjegyzék

[Feladatkiírás 2](#_Toc80443487)

[Tartalmi összefoglaló 3](#_Toc80443488)

[Tartalomjegyzék 4](#_Toc80443489)

[**Bevezetés 5**](#_Toc80443490)

[**1. Téma elméleti háttere 5**](#_Toc80443491)

[1.1. Digitális képfeldolgozá 5](#_Toc80443492)

[1.2. Diagramok 5](#_Toc80443493)

[**1.2.1. Diagramok, mik azok, miért van rájuk szükség 5**](#_Toc80443498)

[**1.2.2. E - K diagram 5**](#_Toc80443499)

[**2.Használt eszközök 6**](#_Toc80443505)

[2.1 Python. 6](#_Toc80443506)

[2.2. OpenCV 6](#_Toc80443507)

[2.3. Numpy 6](#_Toc80443508)

[2.4. PyTesseract 6](#_Toc80443507)

[2.5. PyQt6 6](#_Toc80443508)

[2.6. Drawio 6](#_Toc80443508)

[**3. Diagram elemek felismerése és XML generálás 5**](#_Toc80443495)

[3.1. Elemek és kapcsolatok osztályszintű leírása 5](#_Toc80443496)

[3.2. Előfeldolgozás 5](#_Toc80443497)

[**3.2.1. Képek előkészítése 5**](#_Toc80443498)

[**3.2.2. Hibajavítás 5**](#_Toc80443499)

[3.3. Alakzatfelismerés 5](#_Toc80443497)

[**3.3.1. Hibaellenőrzés 5**](#_Toc80443499)

[3.4. Kapcsolatok keresése 5](#_Toc80443497)

[**3.4.1. Elemek eltávolítás és képtisztítás 5**](#_Toc80443498)

[**3.4.2. Vonalak keresése 5**](#_Toc80443499)

[**3.4.3. Vonalak elemekhez kapcsolása 5**](#_Toc80443498)

[**3.4.4. Validáció 5**](#_Toc80443499)

[**3.4.5. Komplex vonalak 5**](#_Toc80443498)

[**3.4.6. Nyilak definiálása 5**](#_Toc80443499)

[**4. Vizuális felület és szövegfelismerés 5**](#_Toc80443501)

[4.1. Vizuális felület generálása. 5](#_Toc80443502)

[4.2.Szövegkeresés 5](#_Toc80443503)

[4.3. Gyenge elemek és kulcsok. 5](#_Toc80443504)

[4.4. XML fájl készítése 5](#_Toc80443497)

[**5. Eredmények és hibák 5**](#_Toc80443501)

[5.1 Alkalmazás használata. 5](#_Toc80443502)

[**5.1.1. Főablak és szabályok 5**](#_Toc80443498)

[**5.1.2. Szövegfelismerő ablak és felhasználói javítás 5**](#_Toc80443499)

[5.2.Elért eredmények 5](#_Toc80443503)

[5.3.Lényegesebb hibák és megoldások 5](#_Toc80443503)

[**6. Összegzés 5**](#_Toc80443501)

[Irodalomjegyzék 7](#_Toc80443509)

[Nyilatkozat 8](#_Toc80443510)

[Köszönetnyilvánítás 9](#_Toc80443511)

# BEVEZETÉS

Programozóként az első lépes a kódoláshoz, vagy egy projekthez, egy terv kialakítása. Itt jönnek elő a diagramok. Használatuk segít vizualizálni a folyamat elemeit, azokat kialakítani, megtervezni, és javítani. A diagramok arra is szolgálnak, hogy egyszerű leírást adjanak akár a velünk dolgozóknak, akár olyanoknak, akik nincsenek tisztában az egész eddigi munkafolyamattal. Természetesen ehhez kell egy bizonyos megértés a diagram felépítéséhez, annak elemeihez és azok jelentéséhez. Mielőtt elkezdnénk készíteni egy diagramot, fennállhat a kérdés, hogy ezt milyen eszközzel is tegyük. Csináljuk kézzel esetleg, vagy erre kifejlesztett rajzoló programmal? Ha rajzoló programmal készítjük, onnan kinyomtatni egyszerű már, illetve a későbbi szerkesztés is sokkal könnyebb, de fordítva nem ez a helyzet. Szakdolgozatom célja ennek a folyamatnak a megkönnyítése, automatizálása.

Dolgozatomban az egyed - kapcsolat diagrammokra fókuszáltam, amiknek a főbb célja az az elkészítéséhez adatbázis modellezése átlátható formában, illetve azt vizsgáltam, hogyan lehet E - K diagram elemeit algoritmikusan felismerni.

Amennyiben az adott diagram megfelel a technikai feltételeknek, a program egy PyQt grafikus felületen keresztül feldolgozza a raszteres képet, azt vektorgrafikus rajzoló és szerkesztő program által kezelhető formába hozza. Ebben a felhasználónak is szerepe van, bizonyos beállításokat neki kell elvégezni.

A dolgozatom első részében felvázolom a téma elméleti háttereit, konkrétan a digitális képfeldolgozás alapjait, és az E - K diagramok felépítését. A második részben, bemutatom a program megíráshoz használt eszközöket, majd az ezt követő részben magát a kódot tárgyalom, azon belül részletesen a képfeldolgozást és annak segédeszközeit. A képfeldolgozási folyamat három részben fogom tárgyalni: előfeldolgozás, alakzatfelismerés és a kapcsolatkeresés. Ezek a lépesek során kinyert információk alapján már legenerálható a vektorgrafikus fájlt.

A negyedik rész szól a felhasználói felületről, illetve az ott megadható beállításokról. Az ötödik részben az használati útmutatóról, az érdekesebb hibákról, amik előjöttek az alkalmazás készítése során, illetve az elért eredményekről szól. Kitérek arra is, hogy milyen hibák jöttek elő a szakdolgozatom kidolgozása alatt. Szerintem az eredmények áttekintése ezek nélkül nem teljes. Végül a hatodik fejezeben összefoglalnám a teljes folyamatot.

**1. A TÉMA ELMÉLETI HÁTTERE**

Ebben a fejezetben a digitális képfeldolgozás alapjait ismertetem, majd a diagrammok, azon belül is az E -K diagramnak a szerepét, elemeit, felépítését, és egyéb tulajdonságait tárgyalom.

**1.1. Digitális képfeldolgozás**

Ha látunk egy képet magunk elött, az agyunk megpróbálja azt értelmezni. Mit ábrázol, milyen színek jelennek meg benne, az egyes részeket külön és akár egy nagyobb egész részeként is felismerni, mondhatni információt nyerni a képből. A digitális képfeldolgozás ennek a folyamatnak a számítógép által végzett megfelelője. Több feladatot elvégezhetünk a használatával: minőség javítás, szegmentálás, jellemzők kinyerése, információ gyűjtés, kép analízis. Bármelyiket is válasszuk, több mint valószínű, hogy a folyamat első lépése az előfeldolgozás.

Ameddig az emberek képesek logikai következtetéseket vonni, elemetek akár rossz minőség ellenében is értelmezni, ugyan ez nehezen megfogalmazható kód formájában. A képek képmátrixként vannak reprezentálva, ami egy kettő vagy több dimenziós tömb, amin különböző műveleteket végzünk. Ezeknek a célja a kép javítása, mondjuk zajszűrés, vagy a kontrasztok kiegyenlítése, hogy a kép értelmezhetőbb legyen, a folyamat eredménye ne legyen minőség béli hibák miatt nem megfelelő.  
 Következő lépés maga az elemzés. Attól függően, hogy mi a cél, különböző módon haladhatunk tovább. Képesek vagyunk a kép elemeit csoportosítani valamilyen feltétel alapján (pl.: szín vagy forma), vagy éleket, összefüggő elemeket keresni, akár részekre osztani, szegmentálni a kiindulási képet. Ezek az eljárások kombinálhatóak, sorban elvégezhetőek.

Amennyiben tudjuk mi a cél, csak meg kell határozni a szükséges lépéseket, azokat helyes sorrendben elvégezni. Ez nem mindig sikerülhet elsőre, de mivel ez egy vizuális jellegű téma, a részeredményeket bármikor megjeleníthetjük magunk elött, azokból levonhatunk következtetéseket, és így javíthatjuk a kódunkat, hogy megfelelő eredményt kapjunk.  
 A digitális képfeldolgozás implementációjára több példa is van, több programozási nyelvnek megvan a saját modulja hozzá, mindegyik saját előnyeivel és hátrányaival. Én az OpenCV csomagot használtam, hogy elkészítsem a programomat.

**1.2. Diagramok**

A diagramok fő célja az adatok, információk vizuális reprezentációja, hogy az értelmezhetőbb legyen, akár olyanok, akik nincsnek otthon az adott témában is megérthessék az adott pontokat.

Vegyünk példának mondjuk egy Excel táblázatot, amiben fel van jegyezve egy cég kiadásai, és bevételei, azok lebontva részekre. Akik ezt a táblázatot készítették átlátják a sok nyers adatot, de lehet mások, akik nem jártassak ezen a területen elvesznek benne. Ha viszont van egy diagram, ami egy tömörebb, egyszerűbb képet ad a rengetek információról, akkor könnyebben értelmezhető lesz az adathalmaz.

Számos különböző diagram típus létezik, oszlop és kőr diagramok, vagy programozó környezetben inkább használt példák, mondjuk osztály, csomag és Egyed – kapcsolat diagramok.

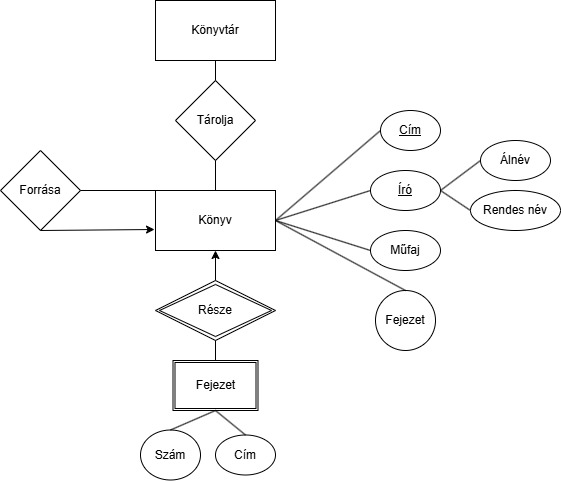
1.2.1. E – K diagram

Egyed – kapcsolat, vagy E – K diagram röviden, ennek a diagram típusnak a fő célja az adatbázisok logikai modelljének elkészítése, reprezentációja. Ez segít az adatbázisok elkészítésében, annak javításában, ha módosítani kell, annak megtervezésében. Illetve, ha meg akarjuk osztani valakivel az adatbázisunkat, egyszerűbb megosztani annak diagramját átadni, hogy ne magából az adattáblákból keljen kitalálni, mi és hogyan kapcsolódik más elemekhez. Három fő eleme van az E – K diagramoknak, *egyedek*, *kapcsolatok* és *attribútumok*. Az 1.1-es ábrán láthatunk egy példát.

Az egyedek, vagy entitások rendelkeznek jellemzőkkel, tulajdonsággokkal, és reprezentálnak egy táblát az adatbázisunkban. Ezeket téglalappal jelöljük, és bele írjuk a nevüket. Például, egyed mondjuk egy könyve, aminek van írója, műfaja, oldalszáma.

A kapcsolat, meghatározza a viszonyt két vagy több egyed között. Ennek a jelölése egy vonal az egyedek között, amin egy rombusz van, abba bele írva a kapcsolat neve. Amennyiben kapcsolat vissza mutat önmagára, akkor *rekurzív kapcsolatról* beszélünk, mondjuk egy könyve végén fel vannak jegyezve a források, akkor az egy rekurzív kapcsolat lehet, hiszen a források lehetnek ugyan úgy könyvek.

Mind, egyedeknek, mind kapcsolatoknak vannak *attribútumaik*, amik a tulajdonságok (könyve: cím, író, műfaj). Attribútumok jelölése egy ellipszis, amiben bele van írva a nevük. Ha ez a név alá van húzva, akkor az a *kulcs*, ami képes egyértelműen azonosítani az egyedet. A kulcs nem minden esetben egy tulajdonság, lehet több is. Csak egyedül az író, vagy a cím alapján nem biztos, hogy melyik könyvre gondolunk, de a kettő együtt meg tudja határozni. Ezen felül lehetnek *összetett attribútumaink* is, olyan tulajdonságok, amik rendelkeznek további tulajdonságokkal (Egy írónak lehet írói álneve, illetve a rendes, anyakönyvi neve is)

Nem minden egyednek van önmagában értelme. Lehet, hogy nem létezik olyan adat, amely azt önmagában meghatározza az entitást, másszóval nincs kulcsa. Ezeket hívjuk *gyenge egyednek*, jelölésük egy dupla vonalas téglalap. Ilyenek esetén lennie kell a gyenge egyed és egy hozzá kapcsolható nem gyenge egyed között, egy *meghatározó kapcsolatnak*. Ennek jele az összekapcsoló vonalon lévő dupla vonalas rombusz, illetve egy nyíl a vonal végén, ami a nem gyenge entitásra mutat. Egy könyvnek vannak fejezei, azoknak viszont magunkban nincs értelmük, mert a könyv részei, és anélkül nincs értelmük.

**1.1. ábra** – E–K diagramra egy példa

Végül, habár nem olyan sokszor van rá szükség, léteznek *specializáló kapcsolatok* is. Ezeknek a szerepük a hierarchia jelölése egyedek között. Jele egy háromszög a vonalon, aminek a csúcsa a főtípus felé mutat. Példa: Egy könyve a főtípus, annak van írója, oldalszáma. Egy kisebb csoport ezen belül mondjuk egy tankönyv, ami az előbb említett attribútumok mellett rendelkezik saját, egyedi tulajdonsággal, mondjuk szómagyarázattal, de ugyan úgy könyv.

**2. Használt eszközök**

Most, hogy már ismerjük a téma hátterét, térjünk rá pontosan milyen eszközök kellettek a program elkészítéséhez.

**2.1. Python**

A személyes preferenciámat leszámítva, a Python-t egyszerűsége (egyik fő előnye az, hogy az angol nyelvre épül, és ezért könnyen értelmezhető) és sokoldalúsága miatt választottam. Mivel nem voltam benne eleinte biztos, milyen eszközökre lesz szükségem, így azt is figyelembe kellett vennem, hogy olyan programozási nyelvet válasszak, ahol nem fordul elő, hogy egy, a témához szükséges elemet, a választott nyelv nem támogat.

Rengeteg különböző területre kiterjedt mára a Python. Legyen szó chatbotok és AI programok készítéséről, grafikus felületek létrehozásáról, vagy képfeldolgozásról, ezek mindegyikét támogatja. Ezt programcsomagok, modulok és függvény könyvtárak formájában érhetjük el. Ezeket legtöbb esetben csak telepíteni kell, aztán importálni és készen is vagyunk, nagyobb csomagok esetén lehet bonyolultabb a folyamat. Egyetlen hátránya ezeknek, hogy egy részük nem Python-ban van leprogramozva, hanem mondjuk C vagy C++-ban, és nem lehet őket egyszerűen átírni.

Azt kell még tudnunk, hogy ez egy interpretált nyelv, azaz nincs szükség fordítóprogramra, viszont ezért cserépe a futásidő lassabb lehet a fordított nyelvekhez képest. Objektum orientált programozásra lett kitalálva, de nincs megszabva, hogy csak így lehessen kódolni benne.

Összegezve, ez egy igen rugalmas, egyszerűen tanulható és használható programozási nyelv, ami számomra tökéletes volt a szakdolgozatomhoz.

**2.2. OpenCV**

Az *OpenCV* (Open Source Computer Vision Library) egy nyílt forráskódú könyvtár, ami magába foglal több száz számítógépes látás algoritmust [4]. Eredetileg C-ben írták meg, de később áttértek C++ nyelvre. Ezt a függvény könyvtárat mind Java, mind Python környezetben tudjuk használni, nagyjából ugyan olyan módon, de vannak minimális eltérések a nyelvek sajátosságai miatt, én az utóbbit választottam.

A képek OpenCV-ben képmátrixként, egy kettő vagy több dimenziós tömbként vannak reprezentálva. A (0,0) koordinátájú pont a bal felső sarok, első szám az adott pont x (vízszintes) tengelyen lévő koordinátája, második az y (függőleges) tengelyé. Mindegyik képpont rendelkezik *intenzitással.* Ha a kép szürke árnyalatos akkor ez egy szám 0 és 255 között, ahol 0 a fekete, 255 a fehér. Amennyiben színes képről beszélünk, akkor mindegyik színcsatornának saját intenzitása van kék, zöld, piros sorrendben, hasonló értékek között. Minél nagyobb az érték, annál világosabb a szín.

Egy számomra fontos funkció, a *szegmentálás*. Ez a kép részekre bontása, egy objektumot alkotó képpontok meghatározására szolgáló folyamat. [1] Több szempont szerint el lehet végezni, mondjuk egy területen lévő hasonló színek szerint, vagy, amit én is használtam, képen látható élek szerint. Ezek a részek alapján aztán további műveleteket lehet végezni, de egyesesetekben ehhez nem elég az OpenCV.

**2.3. NumPy**

A *NumPy* a Python tudományos számítsa technikai alapcsomagja. [5] Egy nyílt forráskódú, ingyenes, és egyszerűen használható programcsomag, amit segít a bonyolultabb számítások és tömb operációk gyorsabb elvégzésében.

Amennyiben OpenCV-t használunk, előbb-utóbb szükségünk lesz NumPy-ra is. Ez azért van, mert a képmátrixokon végzett műveleteket elvégzését megkönnyíti, illetve javítja az időigényüket, mert C-ben van megírva, ami gyorsabb, mint a natív Python-hoz képes. Például használatával nem kell kézzel maszkokat, egyfajta mátrixot a képműveletekhez, kézzel megírni, hanem elég, ha megadjuk a mátrix dimenzióit, és generál nekünk egyet. Ezen felül több képfeldolgozási operáció a csomag része, például élsimítás, konvolúció, Fourier-transzformáció.

**2.4. PyTesseract**

A *PyTesseract* egy optikai karakter felismerő (OCR) eszköz Python-hoz. Azaz, a fő szerepe, hogy felismerje és „leolvassa” a szöveget a vizsgált képről [6].

Maga a PyTesseract az eredeti, Google által fejlesztett, Tesseract karakter felismerőnek egy csomagoló osztálya, ami lehetővé teszi, hogy Python nyelvel kompatibilis legyen. Használatával kepések vagyunk képekből szövegek kinyerni, amennyiben a támogatott kép típusok egyike (jpeg, png, gif stb.).

Én arra használta, hogy a diagram elemeiből kiolvassam az egyedek, és a kapcsolatok neveit, illetve az attribútumokat, hogy később hozzá tudjam adni a végeredményben előállított vektorgrafikus képhez.

A karakterfelismerés pontossága több tényezőtől is függ, mint például a kép minősége, felbontása, a szöveg és a háttér kontraszt béli különbsége, de a legfontosabb tényező maga az olvashatósága a kézírásnak. Mivel az íráskép emberenként változó, a karakterfelismerés pontossága sem konstans.

A 4.2. fejezeben fogom tovább tárgyalni a szövegkeresés, annak az általam használt implementációját bemutatni.

**2.5. PyQt6**

A *Qt* maga, egy gyűjtemény, ami számos C++ keresztplatform alapú könyvtárat foglal magába. Használatával képesek vagyunk mobil és számítógépes alkalmazásokhoz vizuális felületet készíteni, illetve elérni más, külső elemeket hozzá: helymeghatározás, multimédia, NFC, Bluetooth, Chromium alapú webböngészők [7].

A *PyQt6* egy kötéskészlet, aminek a segítségével a Qt v6-nak (6. verzió) az elemei Pythonban is tudjuk használni. Ezáltal lehetővé teszi, hogy a Python is működjen alternatívaként C++ mellett alkalmazás vizuális felületének készítésére, akár iOS vagy Android rendszerekhez is.

A GUI részt a programomhoz PyQt6 segítségével készítettem, mert flexibilis és egyszerű. Van grafikus felület is hozzá, a *Qt Designer*, de nem éreztem szükségesnek, hiszen nem használtam animációk vagy külső elemeket, ezért az elemek csak a kódban vannak definiálva, a kód futtatása során lesznek legenerálva.

A grafikus felület felépítését, különböző részeit, azok szerepét és használatukat, az Alkalmazás használata fejezetben fogom részletessebben tárgyalni.

**2.6. Drawio**

A *Drawio* egy alkalmazás, ami leegyszerűsíti a diagramok készítését, szerkesztését. Elérhető mind webes felület (draw.io), mint letölthető alkalmazás formájában.

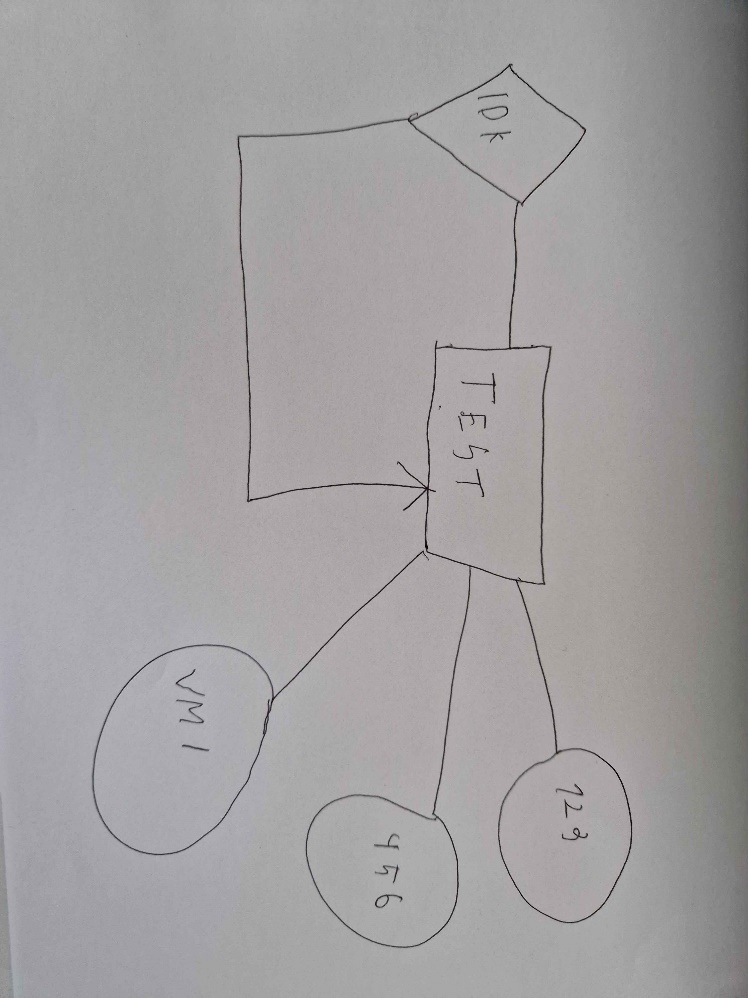
A vektorgrafikus képeket, amik a program sikeres lefutása alatt keletkeznek, magukban nem, lehet megnyitni. Ezt oldja meg a Drawio, am i képes ezeknek a fájloknak a kezelésére. Ez eredmények .drawio kiterjedéssel rendelkeznek, hogy a gond nélkül képes legyen a program vagy a web felület feldolgozni, de igazából ezek .xml fájlok, és annak struktúráját, szintaxisát követik.

Több lehetőség is rendelkezésre állt a vektor képek megtekintéséhez, az eredmény szerkeszthetőségének tesztelésére (például: Dia Diagram Editor), de mivel Drawio-t már ismertem, ezért ezt választottam.

**3. DIAGRAM ELEMEK FELISMERÉSE ÉS XML GENERÁLÁS**

Ebben a fejezetben ismertetem a program azon elemeit, amelyek elvégzik a diagram előkészítéstét, leolvassák róla az adatokat, és azokból létrehozzák a céleredmény vázát.

A 3.1-es ábrán fogok szemléltetni a folyamatot, annak változásait láthatjuk a fejezet folyamán.



**3.1. ábra** – kézzel rajzolt E - K teszt diagram

**3.1. Elemek és kapcsolatok osztályszintű leírása**

A diagram elemeit kettő részre osztottam fel, az alapján, hogy milyen módszert használtam a felismerésükre, geometriai síkidomok és vonalak, amelyek összekapcsolják azokat egymással. Hogy egyszerűbben tudjam kezelni az kinyert adatokat, és tudjam őket külön-külön, egyesével feldolgozni, készítettem ezekhez kettő osztályt, *Element* és *Line* néven.

Az Element osztály reprezentálja az alakzatokat, mint a háromszög, a négyzetek, vagy a körük. Az idadattag azonosítja az egyes elemeket, a x és y tárolja az elem bal felső sarkának koordinátáit, és a width és height meghatározza a dimenzióit, végül pedig a shape megmondja milyen alakzat. Ezek kellenek az alap működéshez, ezen felül van még a text, ami a bele írt szöveg, az underline, hogy aláhúzott-e a szöveg vagy nem, és a double, ami azt jelzi, hogy dupla vonalas-e a kőrvonala.

A Line osztály adattagjai: id, x1, y1, x2, y2, connection1, connection2, line\_type, pointing\_at. Az a koordináta párok, x1 és y1, valamint x2 és y2, a vonal két végpontját tárolják. Connection1 és connection2 egy-egy id-t tárol, elnevezés szerint annak az elemnek az azonosítóját, ami ahhoz a végponthoz legközelebb van. A line\_type lehet Line (egyszerű vonal), Arrow (nyíl), vagy Connector (olyan vonal, ami más vonalakat köt össze). Ha a típus nyíl, akkor a pointing\_at annak az elemnek az id értékét tárolja, amire mutat a nyíl, különben -1.

Minden adattag rendelkezik getterrel és setterrel, hogy könnyel el lehessen érni őket, illetve, Python *OOP* (Objektum Orientált Programozás) szabvány szerint, konstruktorral, hogy lehessen őket példányosítani.

**3.2. Előfeldolgozás**

Mielőtt elkezdődhetne a képfelismerés elő kell készíteni a képet, hogy a folyamat gond nélkül végbe menjen és a diagram elemeit helyesen felismerhessük.

**3.2.1. Képek előkészítése**

Első lépésként átkonvertáltam a képet színes formátumból szürkeárnyalatosba, hogy a színekkel ne keljen foglalkozni, majd ezt követően meghatároztam a kép átlag intenzitását. Ezt az értéket használva, felállítottam egy küszöbértéke és amelyik pixel ez alatt van fehérre, ami felette azt feketére váltok. Tesztelés során arra jutottam, hogy maga az átlag érték nem jó küszöbérték, ezért a program mindig egy 0.9-es szorzót alkalmaz rajta, és az így számolt értéket használja. Így létrejön egy maszk, ahol a diagram elemei fehérrel, a háttér feketével jelenik meg.

Ami a maszkon fehér, illetve a fehér vonalak által bezárt területeket kitöltöm feketével az eredeti képen. Ezt morfológiai nyitást végzek, hogy eltávolítsam a kisebb zajokat, és a vonalakat, amelyek összekötik az elemeket. A testek nem fognak eltűnni, mert a kitöltés miatt meg lettek vastagítva.

Az eredmény egy fekete-fehér kép lesz, amelyen csak az alakzatok szerepelnek. A 3.2-es ábrán láthatjuk ezt.



**3.2. ábra** – Kapcsolat mentes diagram

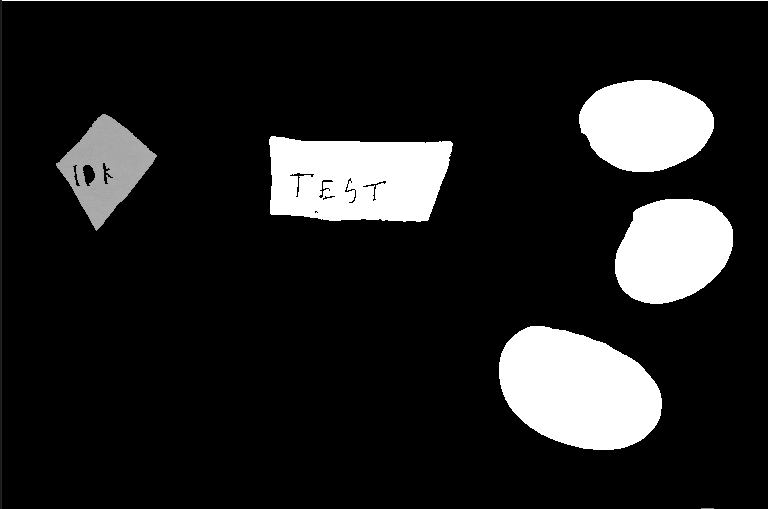
**3.2.2. Hibajavítás**

A 3.2-es ábrán láthatunk kettő hibát. A kép tetején egy nagyobb hamis kontúr, zaj szerepel. Ezzel nem kell egyelőre foglalkoznunk, ez eredeti (3.1.ábra) tetején is láthatjuk. Vagy a kép készítésekor jött létre, vagy a rajzlap hibája, nem fog lényegesen bezavarni a folyamatba, de ha van rá lehetőség kerüljük el az ilyeneket.

A nagyobb hiba az, hogy egyes testek és vonalak összekötése is ki lett töltve, és egy hamis alakzatot alkottak. Észrevehetjük azonban, hogy ez a hibás alakzat jelentősen

nagyobb, mint társai. Ezt a tulajdonságot felhasználva meg tudjuk oldani a problémát.

A 3.2. ábrát maszként használva, rávetítem a kiindulási képre, és egy kontúr kereső függvény, segítségével megvizsgálom az összeset. A méret béli különbséget használva, elsőnek egy fix határértéket, majd később ez előző, helyesnek vélt elem méretét használva keresek nagyobb eltéréseket. Ha ilyet találok, akkor azon belül elkezdtek keresni olyan kontúrokat, amik egy nem túl kicsi területet (azaz zajt) határoznak meg. Ezeknek a kivételével a vizsgált területet kitöltöm feketével, így csak a helyes elemek maradnak láthatóak, és behelyezem az így javított területet a hibás helyére.



**3.3. ábra** – Hibajavított kép csak az alakzatokkal

A 3.3. ábrán láthatjuk az eredményt. Itt már csak a helyes elemek szerepelnek, és még a zaj is eltűnt a kép tetejéről. Ezen már gond nélkül kereshetünk alakzatokat.

**3.3. Alakzatfelismerés**

Az előző lépésben elkészített, zavaró elemektől mentes képen elsőnek elvégzek egy *Gaussi zajszűrést*, hogy az alakzatok élei egyenletesek legyenek. Erre azért van szükség, mert a kézzel rajzol alakzatokban lehetnek kiugró pixelek, és ez hibához vezethet az alakzatok felismerésében.

Ezt követően, egyesével végig megyek a kontúrokon. Ha a vizsgált terület mérete túl kicsi, akkor zajnak mondható, is kihagyjuk. Az *approxPolyDP* függvény az eredeti kontúrból készít egy pontsorozatot, ami közelíti az eredetit, de kevesebb pontból áll, ezáltal egyszerűbb rajta dolgozni, és pontosabb eredményekhez vezet [8].

Ezt használva elsőnek Euklideszi távolságot számítok a szomszédos pontok között, ezáltal megkapom az oldalak hosszait, majd kiszámítom az összes szöget is.

Ezek az információk alapján már csak be kell kategorizálni, hogy a vizsgált elem milyen alakzat. Ha három oldalt talál a program akkor háromszög, több dolga nincs. Ha az oldalszám négy vagy öt (a zajszűrés sem képes tökéletesen kiszűrni mindent, ezt egy minimális hibahatár) akkor meg kell vizsgálni, hogy rombusz vagy négyzet. Ha a szögek közel vannak derékszöghöz, akkor négyzetnek nyilvánítja. Amennyiben ez nem teljesül, megnézi, hogy az egymással szemben lévő szögek megegyeznek-e, ha igen akkor feljegyezi, hogy rombusz. Amennyiben egyik kritérium sem teljesült, akkor kap egy ideiglenes címkét, és később más módon határozza meg, a típusát. Amennyiben ötnél több oldala van, akkor ellipszis.

Végül létrehoz egy új Element osztályú egyedet, abban eltárolja az alakzat típusát, x és y koordinátáit, szélességét, magasságát. Ezt hozzáadja egy tömbhöz, hogy lehessen később vele dolgozni.

**3.3.1. Hibaellenőrzés**

Habár szerkezetileg később következeik logikailak ehhez a részhez tartozik még a hibás alakzat felismerés javítása is.

A tesztek folyamán megfigyeltem, hogy a négyzet és a rombusz felismerése és megkülönböztetése, problémát okozhat egyes, szélsőséges esetekben. Habár ritka, de a kézi rajzokból adódó hibák miatt előfordulhat. Erre azt a megoldás találtam, hogy miután a kapcsolatok is ki lettek következtetve, ezek alapján határozom meg az alakzat típusát.

Végig iterálok az alakzatokat tartalmazó tömbön, és ha az éppen vizsgált elem shape adattagja az ideiglenes címkét viseli, akkor megnézem a hozzá kapcsolt elemek típusát. Például, ha a vizsgált elem rombuszhoz kapcsolódik, az már nem lehet rombusz, hasonlóan négyzet esetében vagy háromszög is csak négyzethez kapcsolódhat. Így az ismeretlen elemek nem fognak problémát okozni, és valószínűleg megfelelően be lesznek csoportosítva.

**3.4. Kapcsolatok keresése**

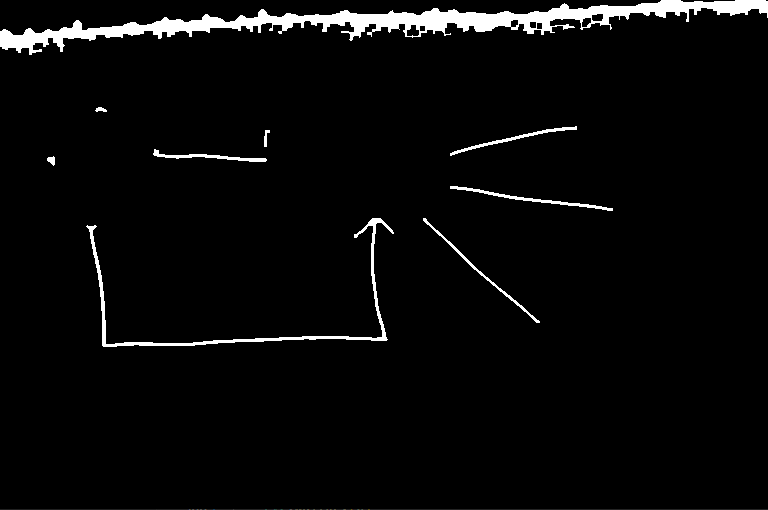
Miután az alakzatokon végzett folyamatok befejeződtek, következik az ezeket össze kapcsoló vonalak feldolgozása. Ehhez szükség lesz az alakzatokból kinyer adatokra is, ezért lett ilyen sorrendben megvalósítva a megoldás.

**3.4.1. Elemek eltávolítás és képtisztítás**

Mivel a testek is vonalakból állnak, és bezavarnának a helyes elemek érzékelésébe, elsőnek eltávolítom őket. Az előző megtalált elemek koordinátait, szélességüket és magasságukat használva, az eredeti kép ezen területeit kivágom és fehér színnel helyettesítem. Ezt követően az egész képen, az eredeti intenzitási átlagot határértékként használva mindent feketére vagy fehérre állítok.

Emellett a képen előfordulhatnak hibás pixelek, sötét foltok miatt keletkezett kontúrok, vagy az alakzatok eltávolítása során megmaradt apró pontok. Ezek eltávolítására elsőnek *adaptív küszöbölést* végzek. Minden pixelre ki lesz számolva egy lokális küszöbérték a környezete alapján. Mivel a Gauss féle módszert használtam, a vizsgált pixeltől való távolság is számít az érték meghatározásában. Ha a kiszámított érték meghaladja a küszöbértéket fekete lesz, ha nem akkor fehér.

Ez nem tüntet el minden zajt, ezért még egy *morfológia zárást*, aminek a célja, hogy a kisebb pontokat a fehér területeken belül eltüntessem, majd egy *morfológiai nyitást*, ami pedig a kis méretű fehér pontok eltüntetésére szolgál. A 3.4. ábrán láthatjuk az eredményt.



**3.4. ábra – Kép a testek eltávolítása után**

Ideális estben csak a vonalak maradnak meg nekünk a műveletek elvégzése után, azonban maradhatnak hibák rajta. Jelen esetben ez az eredeti kép készítésekor keletkezett sötét folt miatt van, mint azt láthatjuk a 3.1. ábrán is. Habár ez a hiba nem fog bezavarni a folyamatba, a kiindulási kép tisztasága nagyban megnöveli a sikeres diagram digitalizálás esélyét.

**3.4.2. Vonalak keresése**

Az előző lépés eredményeként keletkezett képen, mivel ideális esetben csak az alakzatokokat összekötő vonalakat tartalmazza, gond nélkül futtatható a vonalkereső algoritmus. Ehhez én *Hough transzformációt* használtam.

A Hough transzformáció egy olyan algoritmus, aminek a segítségével egyenes vonalakat lehet keresni bináris (fekete-fehér) képeken [9]. Ennek be lehet állítani többek között a detektált vonalak minimális hosszát, az az értéket, hogy mekkora távolság esetén számolunk elemeket egy vonalnak, illetve, hogy hány „szavazat” kell az érzékelt elem elfogadásához. Több különböző kép tesztelése után az ideális paraméterezést én úgy találtam, hogy 5-5 pixel volt az első kettő érték, és 10 szavazatra legyen szükség.

Miután ez az algoritmus lefutott, mindegyik érzékelt vonalhoz létre hozok egy Line osztályú objektumot. Ennek a pontos leírása 3.1. fejezetben található.

Azt figyeltem meg tesztelés folyamán, hogy az egyenesek nem minden esetben vannak egyben érzékelve, sokszor szétesnek kisebb töredékekre. Erre megoldásként készítettem egy függvény, ami egyesíti azokat a vonalakat, amelyeknek a végpontjai kellően közel vannak egymáshoz, illetve a szögük hasonló. Miután a függvény lefutott kapok egy listát az összes vonalról és ezzel dolgozok tovább.

**3.4.3. Vonalak elemekhez kapcsolása**

Így, hogy a vonalak adatait ismerem, már csak azt kell tudnom, hogy melyik alakzathoz csatlakoznak. Ennek meghatározására az Euklideszi távolságot számoltam az adott végpont, és a testek pixelei között. A 3.5. ábrán láthatjuk a képletet.



**3.5. ábra – Euklideszi távolság két pont között.**

X1 és Y1 az egyik, X2 és Y2 a másik pont koordinátai.

Beállítottam egy minimális távolságot, majd amennyiben találtam ennék kisebbet ezt felülírtam, és elmentettem az azonosítóját a testnek, amelyhez az érték tartozott. Ha nem volt olyan érték, ami kisebb volt, mint a határérték, akkor az elmentet azonosító ahhoz a végponthoz -1 lett.

**3.4.4. Validáció**

Habár az előző lépésekben próbálkoztam megoldani, hogy hibás kapcsolatok, töredék vonalak, hamis kontúrok ne zavarjanak be a programnak, így is van esély hibás kapcsolatok érzékelésére. Ennek megoldására a egy validációs folyamat fut le, ami ki kell, hogy szűrje a hibák nagyját.

Az első ilyen hiba a rövid töredék vonalakból ered, aminek mind a kettő végpontja ugyan ahhoz a testhez lesz azonosítva. Ez nem megengedett, szóval az ilyen vonalakat félrerakom.

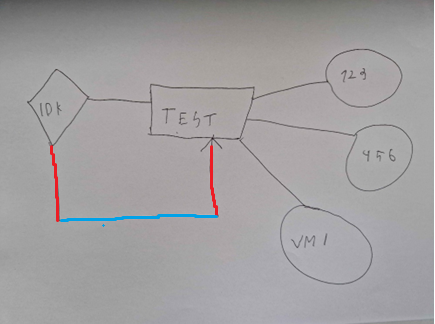
Háromszögek, rombuszok és négyzetek nem kapcsolódhatnak velük megegyező elemhez az E-K diagram szabályai szerint, szóval, ha erre találok példát azt is hibásnak veszem.

Ha az előző két feltétel mellett szerepelnek olyan vonalak a tömbben, amelyek ugyan azt a kettő elemet kötik össze, akkor csak a hosszabbat tartom meg, és a másikat elvetem. Ez a hiba legtöbb esetben a vonalak töredékes érzékelése miatt lehetséges, habár a 3.4.2. fejezetben említett egyesítő eljárás segít, nem minden esetben tünteti el az összeset.

Említettem azt, hogy ugyan olyan alakzatok nem lehetnek összekötve. Erre egy kivétel van, az ellipszis, de az nem minden esetben. Egy ellipszis, azaz attribútum, lehet összetett, azaz rendelkezhet saját attribútumokkal, erre példa a 1.1. ábrán látható. Viszont nem fogadható el egyszerűen két ellipszis kapcsolata sem, hiszen egy kisebb kontúr hiba, vagy egy alakzatból megmaradt töredék miatt lehet hamis kapcsolatot talál a program. Ennek megoldására csak akkor fogadok el vonalat két attribútum között, ha csak egyikük van más típusú alakzathoz kapcsolva, hiszen egy összetett attribútum saját tulajdonságai nem lehetnek csak ahhoz kapcsolva, bármi más azt jelenti, hogy ez egy hiba.

**3.4.5. Komplex vonalak**

Azokat a vonalakat definiálom komplexnek, amelyeknek az egyik végpontja nem alakzatokhoz van csatolva, hanem egy másik vonalhoz. Erre akkor lehet például szükség, amikor egy kapcsolat vissza mutat az alakzatra magára.



**3.5. ábra - komplex vonal**

A 3.5. ábrán pirossal jelölt vonalakt a program gond nélkül megtalálja, és a validáció át is engedi, viszont a kéket megfogja. A piros vonalak egy-egy végpontja csatlakozik egy alakzathoz, a másik viszont nem, így annak a kapcsolatnak az értéke -1 értékkel rendelkezik. Ezt a tulajdonságot felhasználva keresek elsőnek egy ilyen vonalat, aztán hozzá egy másikat, és elmentem azokat végpont koordinátákat, amelyek a -1 értékkel rendelkeznek.

Ez nem minden esetben megfelelő viszont. Ezen a példán csak kettő ilyen vonal van, de ha négy lenne, akkor lehet rossz párosítást használna a program. Ennek megoldására megnézem, hogy a két pont kapcsolata átmenne-e egy teszten, és ha igen akkor keresem másik párt az első koordinátának.

Ha van megfelelő párosom, akkor megnézem vannak-e köztük olyan vonal, amelynek a végpontjai a két koordináta körül vannak. Ha több ilyet találok, vagy külön-külön vonalat találok, akkor elkezdtem ezeket a vonalakt összekötni, amíg létre nem jön egy olyan, amelynek mind két vége megfelelő. Az így elkészített kapcsolatnak létre hozok egy új Line osztályú egyedet, „Connector” típussal, hozzá adom a validált egyenesek listájához, és frissítem a régebbi vonalakat végpontjait, amelyeket összeköt az új.

**3.4.6. Nyilak definiálása**

Nyilak a vonalak végén lehetnek, kapcsolatok esetén, és az elemek hierarchiájának meghatározásához van rájuk szükség.

Azt a tulajdonságot használtam fel elsőnek, hogy nyíl csak annak a vonalnak a végén lehet, amelynek az egyik végpontja rombuszhoz csatlakozik, és csakis a másik végpontnál lehet nyíl, azaz annál amelyik nem a rombuszhoz kapcsolódik.

Gondod csak az okozhat, ha a talált vonal nem direkten egy másik elemhez kapcsolódik, hanem egy másik vonalhoz. Ez esetben egy útkereső algoritmust indítottam, amely a kapcsolatokon végig haladva addig megy, amíg el nem éri a megfelelő végpontot, amelynél nyíl lehet.

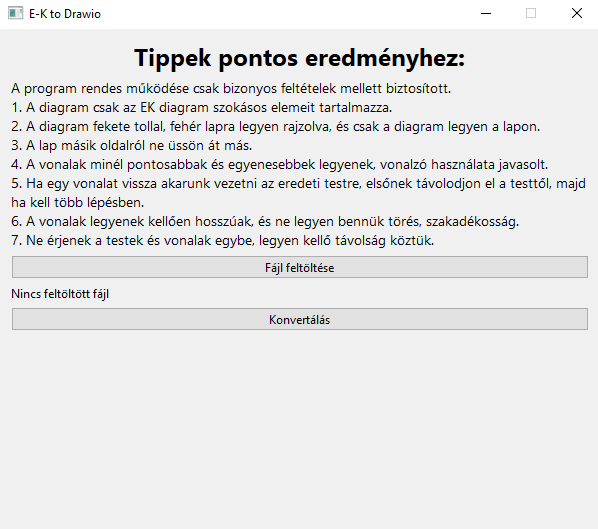
A keresési pont meghatározása után keresek vonalakt, amelyek a nyíl fejét alkothatják ez a pont körül, és elmentem őket. Kiszámolva a szögek különbségé az eredeti egyenes, és a nyílhegy feltételezett részei között, ha ez 25 és 75 között van, akkor elfogadom, mint nyílhegy, és az eredeti egyenes típusát átállítom „Arrow” -ra.

Előfordult egyesesetekben, hogy a vonaltöredezés miatt, a végpont nem pont az alakzat mellett volt, és ezáltal a nyíl részeit nem találta meg, mert túl messze voltak. Ennek megoldására megnézem a vizsgált pont és a hozzá tartozó test távolságát, és ha elég nagy, akkor keresek köztük olyan vonalakt, amelyek a nyilat alkothatják. Ha találok ilyet, akkor ugyan úgy „Arrow” -ra állítom a típust.

**4. VIZUÁLIS FELÜLET ÉS SZÖVEGFELISMERÉS**

Az előző fejezetben tárgyalt folyamatok során megszerzett információk alapján már le lehetne generálni a diagramot vektorgrafikus formában. Ennek ellenére, hogy ez egyrészt egyszerűbb legyen, illetve egyes lépések biztosan jó eredmény adjanak, még szükség van néhány elemre. Ebben a részben ezeket fogom ismertetni.

4.1 Vizuális felület generálása.

****

**4.1.ábra – A fő ablaka az alkalmazásnak**

Az alkalmazás használatához van egy vizuális felület, hogy átláthatóbb, illetve felhasználó barát legyen. Lényegében ezt kettő részre lehet osztani: a főablak, és egy másodlagos, úgymond szerkesztő ablak. Mind a kettő PyQT6 használatával lett készítve, szóval a kódban haszonlóan vannak létrehozva.

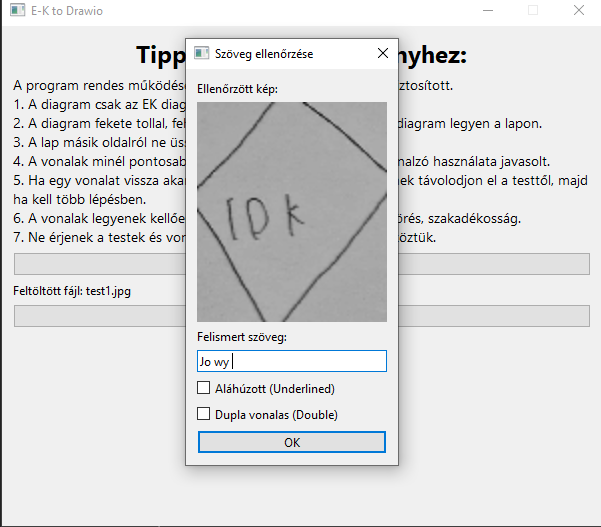
A főablak a *QVBoxLayout* elrendezést használja, amelyt úgy kell elképzelni, mint ha dobozokat helyeznénk egymásra vertikálisan. Így az elemek, a színtérhez adástól függően, egymás alatt jelennek meg. A 4.1. ábrán láthatjuk magát a felületet. A címsor és a szöveges rész alatt van kettő gomb, ezek megnyomására lefut egy-egy előre definiált funkció, a fájlfeltöltés, és a kép konverzió.

Az általam szerkesztő ablaknak nevezet rész akkor jelenik meg, ha elindult a konverziós folyamat. Ezen tudunk különböző opciókat kiválasztani a folyamat utolsó részének, a szövegkeresés, futása alatt.

**4.2.Szövegkeresés**

A szövegek felismerésére és kinyerésére a Pytesseract-ot használtam. Mivel a kézzel írott szöveg eltér emberről emberre az eredmények megoszlóak voltak. Volt olyan szöveg, amit tökéletesen felismer, másokat hibásan vagy egyeltalán nem képes felismerni.

Mivel tudom, hogy írás csak az alakzatokban lehet, így végig iterálok az ezek adatait tároló tömbön, és a területi információk alapján kivágok egy részt az eredeti képből. Hogy pontosabb legyen az eredmény, élesítem a képet, majd meghívom az OCR (Optical Character Recognition) függvényt. Miután ez lefut, az alkalmazás megjeleníti a szerkesztő ablakot. A 4.2. ábrán láthatjuk ezt.



4.2. ábra – A szerkesztő ablak

Hogy a felhasználó tudja melyik elem feldolgozása folyik éppen, megjelenítem az eredeti kép ezen részét. Alatta látható az OCR által talált szöveg. Amennyiben ezt szerkeszteni akarja a felhasználó egyszerűen csak át kell írni. Ezt követően a szöveg hozzá adódik az éppen vizsgált elemhez, hogy később rákerülésén a vektorgrafikus képre.

4.3. Gyenge elemek és kulcsok.

4.4. XML fájl készítése

5. EREDMÉNYEK ÉS HIBÁK

5.1 Alkalmazás használata.

5.1.1. Főablak és szabályok

5.1.2. Szövegfelismerő ablak és felhasználói javítás

5.2.Elért eredmények

5.3.Lényegesebb hibák és megoldások

6. ÖSSZEGZÉS

## Irodalomjegyzék

1. Tanács Attila: Digitális képfeldolgozás jegyzet: <https://www.inf.u-szeged.hu/~tanacs/pyocv/>

Utolsó megtekintés dátuma: 2025. 03. 31.

1. Dr. Németh Gábor, Dr. Kardos Péter, Dr. Bodnár Péter: Adatbázisok gyakorlati jegyzet: <https://www.inf.u-szeged.hu/~gnemeth/adatbgyak/exe/AdatbazisokGyakorlat2020/index.html>

Utolsó megtekintés dátuma: 2025. 04. 04.

1. Python 3.12 dokumentáció: <https://docs.python.org/3.12/faq/general.html#what-is-python>

Utolsó megtekintés dátuma: 2025. 03. 30.

1. OpenCV 4.11 dokumentáció: <https://docs.opencv.org/4.11.0/>

Utolsó megtekintés dátuma: 2025.03.31.

1. NumPy dokumentáció: <https://numpy.org/doc/stable/user/index.html#user>

Utolsó megtekintés dátuma: 2025.04.01.

1. PyTesseract dokumentáció: <https://pypi.org/project/pytesseract/>  
   Utolsó megtekintés dátuma: 2025.04.07.
2. PyQt6 dokumentáció: <https://pypi.org/project/PyQt6/>  
   Utolsó megtekintés dátuma: 2025.04.25.
3. approxPolyDP függvény leírása: <https://docs.opencv.org/4.x/d3/dc0/group__imgproc__shape.html#ga0012a5fdaea70b8a9970165d98722b4c>

Utolsó megtekintés dátuma: 2025.04.13.

1. Hough Line Transform leírása: <https://docs.opencv.org/4.11.0/d9/db0/tutorial_hough_lines.html>

Utolsó megtekintés dátuma: 2025.04.18.

## Nyilatkozat

Alulírott ………………..………… szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Intézet ……………………….. Tanszékén készítettem, …………………….…….…… diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Tudomásul veszem, hogy szakdolgozatomat / diplomamunkámat a Szegedi Tudományegyetem Diplomamunka Repozitóriumában tárolja.

Dátum

Aláírás

## Köszönetnyilvánítás

Mellékletek

A.

B.

C.