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 átkonvertálja az elemeket. 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-t,a kép kezelésre kitalált Python könyvtárat használtam. 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](#_Toc196668882)

[Tartalmi összefoglaló 3](#_Toc196668883)

[Tartalomjegyzék 4](#_Toc196668884)

[BEVEZETÉS 6](#_Toc196668885)

[1. A TÉMA ELMÉLETI HÁTTERE 7](#_Toc196668886)

[1.1. Digitális képfeldolgozás 7](#_Toc196668887)

[1.2. Diagramok 8](#_Toc196668888)

[1.2.1. E – K diagram 8](#_Toc196668889)

[2. Használt eszközök 10](#_Toc196668890)

[2.1. Python 10](#_Toc196668891)

[2.2. OpenCV 11](#_Toc196668892)

[2.3. NumPy 11](#_Toc196668893)

[2.4. PyTesseract 12](#_Toc196668894)

[2.5. PyQt6 12](#_Toc196668895)

[2.6. Drawio 13](#_Toc196668896)

[3. DIAGRAM ELEMEK FELISMERÉSE ÉS XML GENERÁLÁS 13](#_Toc196668897)

[3.1. Elemek és kapcsolatok osztályszintű leírása 14](#_Toc196668898)

[3.2. Előfeldolgozás 15](#_Toc196668899)

[3.2.1. Képek előkészítése 16](#_Toc196668900)

[3.2.2. Hibajavítás 16](#_Toc196668901)

[3.3. Alakzatfelismerés 18](#_Toc196668902)

[3.3.1. Hibaellenőrzés 20](#_Toc196668903)

[3.4. Kapcsolatok keresése 21](#_Toc196668904)

[3.4.1. Elemek eltávolítás és képtisztítás 21](#_Toc196668905)

[3.4.2. Vonalak keresése 22](#_Toc196668906)

[3.4.3. Vonalak elemekhez kapcsolása 23](#_Toc196668907)

[3.4.4. Validáció 23](#_Toc196668908)

[3.4.5. Komplex vonalak 25](#_Toc196668909)

[3.4.6. Nyilak definiálása 27](#_Toc196668910)

[4. VIZUÁLIS FELÜLET ÉS SZÖVEGFELISMERÉS 27](#_Toc196668911)

[4.1 Vizuális felület generálása. 27](#_Toc196668912)

[4.2.Szövegkeresés 29](#_Toc196668913)

[4.3. Gyenge elemek és kulcsok. 30](#_Toc196668914)

[4.4. XML fájl készítése 30](#_Toc196668915)

[5.Alkalmazás használata 32](#_Toc196668916)

[5.1 Főablak használata. 32](#_Toc196668917)

[5.2. Szerkesztőablak használata 33](#_Toc196668918)

[6. ÖSSZEGZÉS 34](#_Toc196668919)

[Irodalomjegyzék 35](#_Toc196668920)

[Nyilatkozat 37](#_Toc196668921)

[Köszönetnyilvánítás 38](#_Toc196668922)

# BEVEZETÉS

A diagrammok célja bizonyos információk, adatok és a köztük lévő összefüggések szemléltetése. Számos különböző területen találkozhatunk velük, például: oktatás, üzleti élet, mérnöki tervezés szoftverfejlesztés. Használatuk segít vizualizálni egy 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 munkamenettel.

Ehhez szükséges az elméleti háttár ismerete a diagram felépítéséhez, annak elemeihez és azok jelentésének megértéséhez. 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 elkészítéséhez adatbázis modellezése átlátható formában. Azt is 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 1. fejezetében felvázolom a téma elméleti háttereit, konkrétan a digitális képfeldolgozás alapjait és az E - K diagramok felépítését. A 2. fejezetben 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 folyamatot a 3. fejezetben tárgyalom, logikai részekre bontva: előfeldolgozás, alakzatfelismerés és a kapcsolatkeresés. Ezek a lépesek során kinyert információk alapján már legenerálható a vektorgrafikus fájlt.

A 4. fejezet szól a felhasználói felületről, illetve az ott megadható beállításokról. Az 5. fejezet a használati útmutató, ami ismerteti az alkalmazás felhasználó felületének felépítését, egyes részeit és a működését. Végül a 6.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, információt kinyerni belőle: mit ábrázol, milyen színek jelennek meg benne, milyen részekből állnak a kép objektumok, milyen szöveg olvasható rajta. A digitális képfeldolgozás ennek a folyamatnak a számítógép által végzett megfelelője. Több feladatot elvégezhetünk a használatával, például: a kép minőségének javítása, képi objektumok szegmentálása, ill. jellemzőinek kinyerése. Bármelyiket is válasszuk, több mint valószínű, hogy a folyamat első lépése az előfeldolgozás.

Ennek a célja a kép javítása, például zajszűrés, vagy a kontrasztok kiegyenlítése, hogy a kép értelmezhetőbb legyen, a folyamat eredménye ne legyen a rosszabb minőségből adódó 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.

A diagramok fő célja az adatok, információk vizuális reprezentációja, egy adott témában oly módon, hogy azt a témához kevésbé értő közönség is könnyen értelmezni tudja.

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

Számos különböző diagramtípus létezik, oszlop és kőr diagramok vagy programozó környezetben inkább használt példák, osztály, csomag és egyed – kapcsolat diagramok.

# 1.2.1. E – K diagram

Egyed – kapcsolat diagram (röviden E – K diagram) fő célja az adatbázisok logikai modelljének elkészítése, reprezentációja. Ez segít az adatbázisok megtervezésében, szükség esetén azok szerkezetének módosításában. Illetve, ha meg akarjuk osztani valakivel az adatbázisunkat, egyszerűbb megosztani, annak diagramját átadni, hogy ne magából az adattáblákból kelljen 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 erre egy példát.

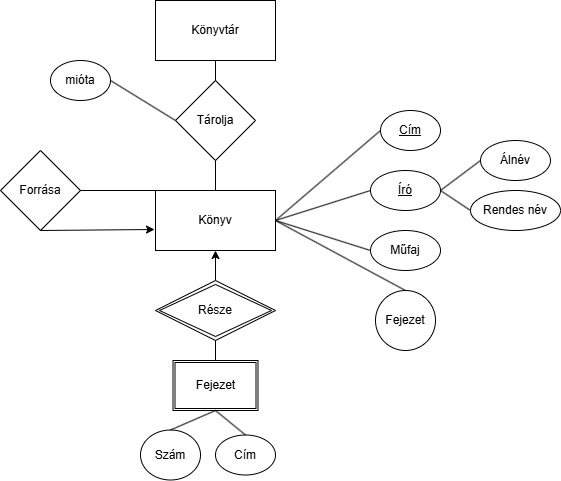
Az egyedek, vagy entitások rendelkeznek jellemzőkkel, tulajdonsággokkal és reprezentálnak egy táblát az adatbázisunkban. Ezeket téglalappal jelöljük és bele írjuk a nevüket. Például, egyed egy könyv, aminek van írója, műfaja, oldalszáma.

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. Egy könyv végén fel vannak jegyezve a források, akkor az egy rekurzív kapcsolat lehet, hiszen a források lehetnek ugyanúgy könyvek.

Az egyedeknek feljegyezzük néhány tulajdonságát, amelyeket attribútumoknak nevezünk (például a könyv attribútumai a cím, író és a műfaj). Fontos megjegyezni, hogy akár a kapcsolatoknak is lehetnek attribútumaik, például: a könyv és könyvtár kapcsolatához felvehetjük a „mióta” attribútumot, arra utalva, hogy mióta található meg az adott könyv az adatbázisban.

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 mindig egyértelmű, hogy melyik könyvre gondolunk, de a kettő együtt meg tudja határozni a helyes választ. 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 önmagában meghatározza az entitást, más szóval nincs kulcsa. Ezeket hívjuk *gyenge egyednek*, jelölésük egy duplavonalas 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 duplavonalas rombusz, illetve egy nyíl a vonal végén, ami a nem gyenge entitásra mutat. Egy könyvnek vannak fejezetei, azoknak viszont önmagukban nincs értelmük, mert a könyv részei és külön 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 lehet egy tankönyv, ami az előbb említett attribútumok mellett rendelkezik saját, egyedi tulajdonsággal, például szómagyarázattal, feladatokkal, de ugyanúgy könyv.

# 2. Használt eszközök

Most, hogy már ismerjük a téma hátterét, térjünk rá pontosan milyen eszközökre támaszkodtam 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, csak nagyobb csomagok esetén lehet bonyolultabb a folyamat. Egyetlen hátránya ezeknek, hogy egy részük nem Python-ban van leprogramozva, hanem például 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 cserébe a futásidő lassabb lehet a fordított nyelvekhez képest. Objektumorientált programozásra lett kitalálva, de nincs megszabva, hogy csak így lehessen kódolni benne. Pár példa más, támogatott programozási paradigmára: Procedurális programozás, funkcionális programozás, Aszinkron/konkurens programozás. Ezeket akár keverhetjük is, hogy elérjük az általunk kitűzött célt.

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 látásalgoritmust [4]. Eredetileg C-ben írták meg, de később áttértek C++ nyelvre. Ezt a függvénykönyvtárat mind Java, mind Python környezetben tudjuk használni, nagyjából ugyan olyan módon, de vannak minimális eltérések a nyelvek sajátosságai miatt. Én most 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]. Ezt több szempont szerint el lehet végezni: egy területen lévő hasonló színek szerint, vagy amit én is használtam, képen látható élek szerint, de egyéb tulajdonságokat használva is lehetőség van a végrehajtásra. Ezek a részek alapján aztán további műveleteket lehet végezni, de egyes esetekben 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 tömboperációk 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 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 karakterfelismerő (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 a Python nyelvvel kompatibilis legyen. Használatával képesek vagyunk képekből szövegeket kinyerni, amennyiben a támogatott képtípusok egyike (jpeg, png, gif stb.).

Én arra használtam, 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 kontrasztbéli különbsége, de a legfontosabb tényező maga a kézírás olvashatósága. Mivel az íráskép emberenként változó, a karakterfelismerés pontossága sem konstans.

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

# 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 az 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 azt most nem éreztem szükségesnek, hiszen nem használtam animációkat 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 is.

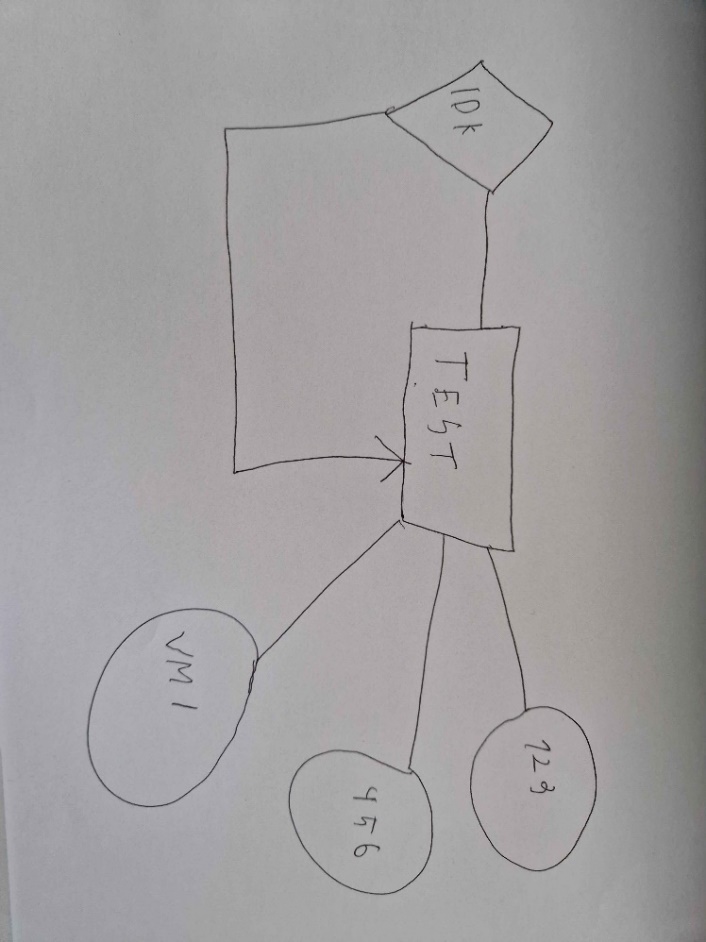
A vektorgrafikus képeket, amik a program sikeres lefutása alatt keletkeznek, magukban nem lehet megnyitni. Ezt oldja meg a Drawio, ami képes ezeknek a fájloknak a kezelésére. Ez eredmények .drawio kiterjedéssel rendelkeznek, hogy gond nélkül képes legyen a program vagy a webfelület feldolgozni, de igazából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

Több lehetőség is rendelkezésre állt a vektorképek 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. Mivel az alkalmazást a Python *OOP* (Objektum Orientált Programozás) szabvány szerint készítettem, függvényekre, illetve osztályokra osztódik fel a kódom.

A 3.1-es ábrán fogok szemléltetni a folyamatot, annak változásait láthatjuk a fejezet során az ábrákon, illetve néhány kódrészletet, amik elvégzik az egyes lépéseket, vagy segítenek azokban.



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

# 3.1. Elemek és kapcsolatok osztályszintű leírása

A diagram elemeit két 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 két osztályt, *Element* és *Line* néven. Ezeket a 3.1. és 3.2. kódrészleten láthatjuk

|  |
| --- |
| class Element:     def \_\_init\_\_(self, id, x, y, width, height, shape, text="", underlined=False, double=False):         self.\_id = id         self.\_x = x         self.\_y = y         self.\_width = width         self.\_height = height         self.\_shape = shape         self.\_text = text         self.\_underlined = underlined         self.\_double = double  ​         def get\_id(self):             return self.\_id  ​         def set\_id(self, value):             self.\_id = value |

3.1. kódrészlet – Element osztály adattagjai, getterre, setterre egy példa

Az Element osztály reprezentálja az alakzatokat, mint a háromszög, a négyzet vagy a kör. Az *id* adattag 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* meghatározza az alakzat formáját. Ezek kellenek az alapműködéshez, ezen felül van még a *text*, ami a beleírt szöveg, az *underline*, hogy aláhúzott-e a szöveg vagy nem, és a *double*, ami azt jelzi, hogy dupla vonalas-e a kör vonala.

|  |
| --- |
| class Line:        def \_\_init\_\_(self, id, x1, y1, x2, y2, connection1, connection2, line\_type,pointing\_at=-1):            self.\_id = id            self.\_x1 = x1            self.\_y1 = y1            self.\_x2 = x2            self.\_y2 = y2            self.\_connection1 = connection1            self.\_connection2 = connection2            self.\_line\_type = line\_type            self.\_pointing\_at = pointing\_at  ​        def get\_id(self):            return self.\_id  ​        def set\_id(self, value):            self.\_id = value  ​ |

3.2. kódrészlet– Line osztály adattagjai, getterre, setterre egy példa

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. A *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 *getter*rel és *setter*rel, hogy könnyen el lehessen érni őket, illetve *konstruktor*ral, 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. Ehhez viszonyítva, felállítottam egy küszöbértéket, amelyik pixel intenzitása ez alatt van fehérre, ami felette azt feketére színeztem.

Tesztelés során arra jutottam, hogy maga az átlagérték 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. Ez után 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. ábrán láthatjuk ezt.



**3.2. ábra** – Kapcsolat mentes diagram

# 3.2.2. Hibajavítás

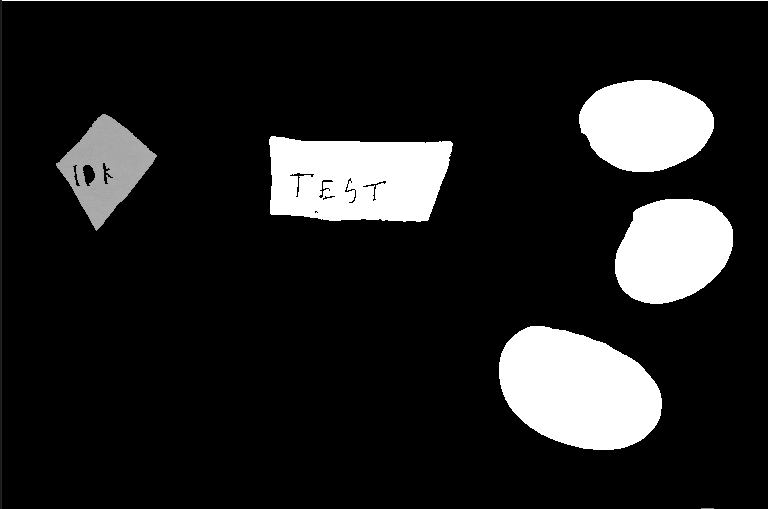
A 3.2. ábrán láthatunk két hibát. A kép tetején egy nagyobb hamis kontúr, zaj szerepel. Ezzel nem kell egyelőre foglalkoznunk, ezt az eredeti (3.1.ábra) tetején is láthatjuk. Vagy a kép készítésekor jött létre, vagy a rajzlap hibája, nem fog lényegesen bezavarni a folyamatba, ameddig nem érintkezik a diagram elemeivel, 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, hogy ez a hibás alakzat jelentősen nagyobb, mint társai. Ezt a tulajdonságot felhasználva meg tudjuk oldani a problémát.

|  |
| --- |
| for contour in contours:  ​     area = cv2.contourArea(contour)  ​     if min\_area < int(area) < max\_area:         cv2.drawContours(masked, [contour], -1, (255, 255, 255), thickness=cv2.FILLED)         x, y, w, h = cv2.boundingRect(contour)         max\_area = area \* 3.2  ​     elif area > min\_area:         x, y, w, h = cv2.boundingRect(contour)           roi = masked[y:y+h, x:x+w]  ​         avrg\_for\_inner = gray.mean() \*0.97  ​         \_, roi\_thresh = cv2.threshold(roi, avrg\_for\_inner, 255, cv2.THRESH\_BINARY\_INV)  ​         inside\_contours, \_ = cv2.findContours(roi\_thresh, cv2.RETR\_TREE, cv2.CHAIN\_APPROX\_SIMPLE)  ​         for inside\_contour in inside\_contours:             inside\_area = cv2.contourArea(inside\_contour)               if 500 < inside\_area < max\_area:                 cv2.drawContours(roi, [inside\_contour], -1, (255, 255, 255), thickness=cv2.FILLED)             else:                 cv2.drawContours(roi, [inside\_contour], -1, (0, 0, 0), thickness=cv2.FILLED)  ​         masked[y:y+h, x:x+w] = roi  ​ |

**3.3. kódrészlet – terület alapú hibajavítás**

A 3.3. kódrészlet, a méretbéli különbséget használva, elsőnek egy fix határértéket, majd később ez előző, helyesnek vélt elem méretét használva keres nagyobb eltéréseket. Ha ilyet talál, akkor azon belül elkezd keresni olyan kontúrokat, amik egy nem túl kicsi területet (azaz zajt) határoznak meg. Ezeknek a kivételével a vizsgált területet kitölti feketével, így csak a helyes elemek maradnak láthatóak és behelyezi 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 *Gauss-simítást*, 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.

|  |
| --- |
| def determine\_shape(self, image):  blurred = cv2.GaussianBlur(image, (5, 5), 0)  \_, threshold = cv2.threshold(blurred, 127, 255, cv2.THRESH\_BINARY)  contours, \_ = cv2.findContours(threshold, cv2.RETR\_TREE, cv2.CHAIN\_APPROX\_SIMPLE)  ​  shapes = []  id = 0  ​  for contour in contours:  if cv2.contourArea(contour) <300:  continue  ​  approx = cv2.approxPolyDP(contour, 0.02 \* cv2.arcLength(contour, True), True)  ​  contour\_points = contour.squeeze()  side\_lengths = []  for i in range(len(contour\_points)):  p1 = contour\_points[i]  p2 = contour\_points[(i + 1) % len(contour\_points)]  side\_lengths.append(cv2.norm(p1 - p2))  ​  angles = []  for i in range(len(approx)):  p1 = approx[i - 2][0]  p2 = approx[i - 1][0]  p3 = approx[i][0]  angle = self.angle\_between(p1, p2, p3)  angles.append(angle)  ​  x, y, w, h = cv2.boundingRect(contour)  ​  shape = "Ismeretlen"  if len(approx) == 3:  shape = "triangle;whiteSpace=wrap;html=1;"  elif 5>= len(approx) >= 4:  ​  if all(85 <= angle <= 95 for angle in angles[:4]):  if abs(side\_lengths[0] - side\_lengths[2]) < 10 and abs(side\_lengths[1] - side\_lengths[3]) < 10:  shape = "rounded=0;whiteSpace=wrap;html=1;"  else:  shape = "rounded=0;whiteSpace=wrap;html=1;"  elif len(angles) == 4 and abs(angles[0] - angles[2]) < 5 and abs(angles[1] - angles[3]) < 5:  shape = "rhombus;whiteSpace=wrap;html=1;"  else:  shape = "vmi4"  elif len(approx) > 5:  shape = "ellipse;whiteSpace=wrap;html=1;"  ​  elem = self.Element(id, x, y, w, h, shape, "")  shapes.append(elem)  id += 1  ​  ​  return image, shapes |

**3.4. kódrészlet – Alakzatok felismerése és kategorizálása**

A 3.4. kódrészlet használatával kategorizálom az alakzatokat. Egyesével végig megyek a kontúrokon, ha a vizsgált terület mérete túl kicsi, akkor zajnak veszem, és kihagyom. 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 (3.1. képlet) számítok a szomszédos pontok között, ezáltal megkapom az oldalak hosszait, majd kiszámítom az összes szöget is.

**3.1. képlet - Euklideszi távolság képlete**

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, ez 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 ellipszisnek számítja.

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 következik, logikailag ehhez a részhez tartozik még a hibás alakzat felismerése és javítása is.

A tesztek folyamán megfigyeltem, hogy a négyzet és a rombusz felismerése és megkülönböztetése problémát okozhat egyes, szélsőséges esetekben. Habár ritka, de ez 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 is, de háromszög is csak négyzethez kapcsolódhat. Így az ismeretlen elemek nem fognak problémát okozni és valószínűleg megfelelően be lesznek csoportosítva.

# 3.4. Kapcsolatok keresése

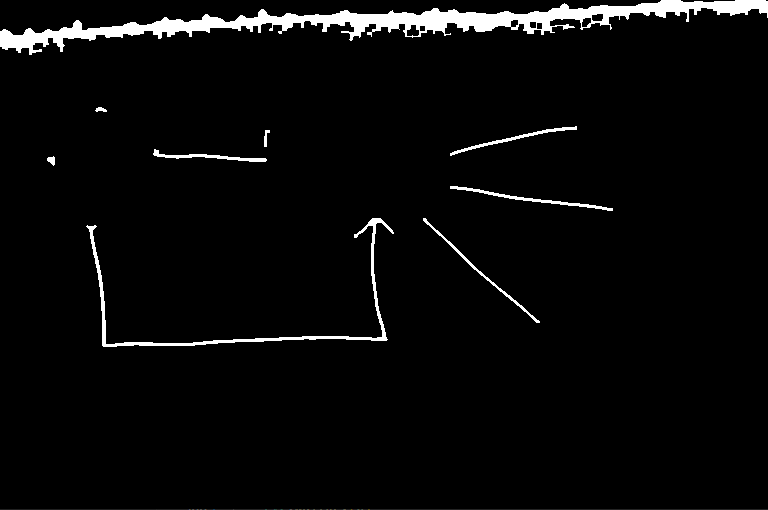
Miután az alakzatokon végzett folyamatok befejeződtek, következik az ezeket összekapcsoló vonalak feldolgozása. Ehhez szükség lesz az alakzatokból kinyert adatokra is, ezért lett ilyen sorrendben megvalósítva a megoldás.

# 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 elvégzek egy *morfológia zárást*, aminek a célja, hogy a kisebb pontokat a fehér területeken belül eltüntesse, 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 digitalizálásának esélyét.

# 3.4.2. Vonalak keresése

|  |
| --- |
| def find\_lines(self, image, shapes):  ​  lines = cv2.HoughLinesP(  image,  rho=1,  theta=np.pi / 180,  threshold=10,  minLineLength=5,  maxLineGap=5  )  ​  lines\_data = []  id = len(shapes)  ​  if lines is not None:  for line in lines:  x1, y1, x2, y2 = line[0]  a\_line = self.Line(id, x1, y1, x2, y2, 0, 0, "Line")  lines\_data.append(a\_line)  id += 1  ​  merged\_lines = self.merge\_lines(lines\_data, len(shapes), 5,15)  ​  return merged\_lines  ​ |

**3.5. kódrészlet – Vonalkereső függvény**

Az előző lépés eredményeként keletkezett képen, mivel ideális esetben csak az alakzatokokat összekötő vonalakat tartalmazza, gond nélkül futtatható a vonalkereső algoritmus. Ehhez *Hough transzformációt* használtam, a 3.5. kódrészleten látható a konkrét megvalósítás.

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 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 paraméterezés az lett, hogy 5-5 pixel volt az első kettő érték és 10 szavazatra legyen szükség.

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

A tesztelés folyamán az látszódott, hogy az egyenesek nem minden esetben lettek egyben érzékelve, sokszor szétestek kisebb töredékekre. Erre megoldásként készítettem egy függvényt, ami egyesíti azokat a vonalakat, amelyeknek a végpontjai kellően közel vannak egymáshoz, illetve a szögük hasonló. Miután a függvény lefutott, kaptam egy listát az összes vonalról és ezzel dolgoztam tovább.

# 3.4.3. Vonalak elemekhez kapcsolása

Így, hogy a vonalak adatait ismerem, már csak azt kell tudnom, hogy melyik alakzathoz csatlakoznak. Ennek meghatározására euklideszi távolságot számoltam az adott végpont és a testek pixelei között.

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 elmentett azonosító ahhoz a végponthoz -1 lett.

# 3.4.4. Validáció

|  |
| --- |
| for line in lines:  valid = True  c1, c2 = line.get\_connection1(), line.get\_connection2()    if c1 == c2:  valid = False  continue    if c1 >= 0 and c2 >= 0 and shapes[c1].get\_shape() == shapes[c2].get\_shape():  shape = shapes[c1].get\_shape()  if shape not in ["ellipse;whiteSpace=wrap;html=1;", "vmi4", "Ismeretlen"]:  valid = False  ​  ​  for vonal in valid\_lines:  con1, con2 = vonal.get\_connection1(), vonal.get\_connection2()  ​  if (c1 == con1 and c2 == con2) or (c1 == con2 and c2 == con1):  ​  current\_length = ((line.get\_x2() - line.get\_x1()) \*\* 2 + (line.get\_y2() - line.get\_y1()) \*\* 2) \*\* 0.5  existing\_length = ((vonal.get\_x2() - vonal.get\_x1()) \*\* 2 + (vonal.get\_y2() - vonal.get\_y1()) \*\* 2) \*\* 0.5  ​  if current\_length > existing\_length:  valid\_lines.remove(vonal)  valid = True  else:  valid = False  ​  if valid:  valid\_lines.append(line) |

**3.6. kórrészlet – Validáció első lépése**

Habár az előző lépésekben próbálkoztam megoldani, hogy hibás kapcsolatok, töredék vonalak, hamis kontúrok ne zavarjanak be a programnak, így is volt esély hibás kapcsolatok érzékelésére. Ennek megoldására egy validációs folyamat fut le, aminek feladata, hogy kiszűrje a hibák nagyját. A 3.6. kódrészleten látható ennek egy része.

Az első ilyen hiba a rövid töredékvonalakból ered, aminek mind a két végpontja ugyanahhoz 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 ugyanazt a két 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.

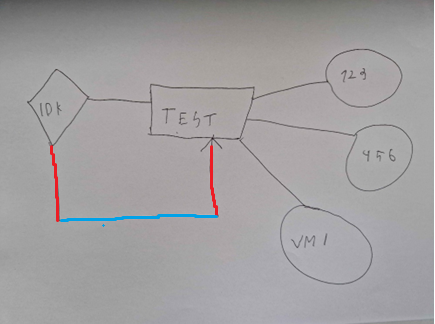
|  |
| --- |
| for line in valid\_lines:  weak1, weak2 = True, True  c1, c2 = line.get\_connection1(), line.get\_connection2()    if c1 >= 0 and c2 >= 0 and shapes[c1].get\_shape() == "ellipse;whiteSpace=wrap;html=1;" and shapes[c2].get\_shape() == "ellipse;whiteSpace=wrap;html=1;":  for vonal in valid\_lines:  if vonal.get\_connection1() == c1 or vonal.get\_connection2() == c1:  if vonal.get\_connection1() >= 0 and vonal.get\_connection2() >= 0:  if shapes[vonal.get\_connection1()].get\_shape() != "ellipse;whiteSpace=wrap;html=1;" or \  shapes[vonal.get\_connection2()].get\_shape() != "ellipse;whiteSpace=wrap;html=1;":  weak1 = False  if vonal.get\_connection1() == c2 or vonal.get\_connection2() == c2:  if vonal.get\_connection1() >= 0 and vonal.get\_connection2() >= 0:  if shapes[vonal.get\_connection1()].get\_shape() != "ellipse;whiteSpace=wrap;html=1;" or \  shapes[vonal.get\_connection2()].get\_shape() != "ellipse;whiteSpace=wrap;html=1;":  weak2 = False    if weak1 or weak2:  true\_valid\_lines.append(line) |

**3.7. kódrészlet – Validáció második része**

Említettem azt, hogy ugyanolyan alakzatok nem lehetnek összekötve. Erre egy kivétel van, az ellipszis, de az sem 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. Minden más azt jelentené, hogy ez egy hiba. A 3.7. kódrészlet ennek a megvalósítása.

# 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 visszamutat önmagára az alakzatra.



**3.7. ábra - komplex vonal**

A 3.7. ábrán pirossal jelölt vonalakat 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 két 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 keresek másik párt az első koordinátának.

Ha van megfelelő párosom, akkor megnézem vannak-e köztük olyan vonalak, amelynek a végpontjai a két koordináta körül vannak. Ha több ilyet találok, vagy külön-külön vonalat találok, akkor elkezdtem ezeket a vonalakat összekötni, amíg létre nem jön egy olyan, amelynek mindkét vége megfelelő. Az így elkészített kapcsolatnak létre hozok egy új *Line* osztályú egyedet, *„Connector”* típussal, hozzáadom 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.

Gondot 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 vonalakat, amelyek a nyíl fejét alkothatják ez a pont körül és elmentem őket. Kiszámolva a szögek különbségét 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 egyes esetekben, 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 megnéztem a vizsgált pont és a hozzá tartozó test távolságát és ha elég nagy, akkor kerestem köztük olyan vonalakat, amelyek a nyilat alkothatják. Ha találtam ilyet, akkor ugyanúgy *„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 lépésre. Ebben a részben ezeket fogom ismertetni.

# 4.1 Vizuális felület generálása.

|  |
| --- |
| def \_\_init\_\_(self):  super().\_\_init\_\_()  ​  self.setWindowTitle("E-K to Drawio")  self.setFixedSize(600, 500)  self.uploaded\_file\_path = None  # Feltöltött fájl nyomon követése  ​  # Fő elrendezés  layout = QVBoxLayout()  layout.setAlignment(Qt.AlignmentFlag.AlignTop)  ​  # Cím középen  title\_label = QLabel("Tippek pontos eredményhez:")  title\_label.setAlignment(Qt.AlignmentFlag.AlignHCenter)  title\_label.setStyleSheet("font-size: 24px; font-weight: bold;")  ​  # Tippek balra  left\_text\_label = QLabel(  "Tippek...."  )  left\_text\_label.setAlignment(Qt.AlignmentFlag.AlignLeft)  left\_text\_label.setStyleSheet("font-size: 14px;")  ​  # Fájlfeltöltő gomb  upload\_button = QPushButton("Fájl feltöltése")  upload\_button.clicked.connect(self.upload\_file)  ​  # Feltöltött fájl nevét mutató címke  self.file\_label = QLabel("Nincs feltöltött fájl")  self.file\_label.setAlignment(Qt.AlignmentFlag.AlignLeft)    # Konvertálás gomb  convert\_button = QPushButton("Konvertálás")  convert\_button.clicked.connect(self.process\_file)  ​  # Elemek hozzáadása az elrendezéshez  layout.addWidget(title\_label)  # Cím  layout.addWidget(left\_text\_label)  # Bal oldali szöveg (szabályok)  layout.addWidget(upload\_button)  # Fájlfeltöltő gomb  layout.addWidget(self.file\_label)  # Feltöltött fájl neve  layout.addWidget(convert\_button)  # Konvertálás gomb  ​  # Beállítjuk az elrendezést az ablakra  self.setLayout(layout) |

**4.1. kódrészlet – A fő ablak kódja**

Az alkalmazás használatához szükség van egy vizuális felületre, hogy átláthatóbb, illetve felhasználóbarát legyen az egész. Lényegében ezt két részre lehet osztani: a főablak és egy másodlagos, úgymond szerkesztőablak. Mind a kettő *PyQT6* használatával lett készítve, tehát 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. kódrészleten láthatjuk ennek a létrehozására szolgáló függvényt. A címsor és a szöveges rész alatt van két gomb, ezek megnyomására lefut egy-egy előre definiált funkció, a fájlfeltöltés és a képkonverzió.

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

# 4.2. Szövegkeresés

A szövegek felismerésére és kinyerésére a *Pytesseract-*ot használtam. Mivel a kézzel írott szöveg eltér minden embernél, az eredmények sikeressége inkonzisztens volt. Akadt olyan szöveg, amit tökéletesen felismert, másokat hibásan vagy egyeltalán nem volt képes felismerni. A 4.2. kódrészlet tartalmazza az általam implementált megoldást.

|  |
| --- |
| def find\_text(self,shapes\_list,gray):  for elem in shapes\_list:  x, y, width, height = elem.get\_x(), elem.get\_y(), elem.get\_width(), elem.get\_height()  ​  x1 = max(0, x + 2)  y1 = max(0, y + 2)  x2 = min(gray.shape[1], x + width - 2)  y2 = min(gray.shape[0], y + height - 2)  ​  roi = gray[y1:y2, x1:x2]  roi = cv2.resize(roi, (0, 0), fx=2.0, fy=2.0)  ​  sharpening\_kernel = np.array([[0, -1, 0],  [-1, 5, -1],  [0, -1, 0]])  ​  sharp = cv2.filter2D(roi, -1, sharpening\_kernel)  ​  text = pytess.image\_to\_string(sharp)  clean\_text = text.replace("\n", " ").replace("\r", " ")  dialog = TextEditDialog(roi, clean\_text)  result = dialog.exec()  # Ez blokkolja a futást, amíg a felhasználó nem nyom OK-t  ​  if result == QDialog.DialogCode.Accepted:  clean\_text = dialog.get\_text()  # Az új szöveg betöltése  underlined = dialog.get\_underlined()  double = dialog.get\_double()  ​  elem.set\_text(clean\_text)  elem.set\_underlined(underlined)  elem.set\_double(double) |

**4.2. kódrészlet – Szövegkereső függvény**

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 szerkesztőablakot.

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. Az „**OK**” gomb megnyomására a szöveg hozzáadódik az éppen vizsgált elemhez, azon belül is a *text* adattaghoz. A kód az itt tárolt adatokat fogja felhasználni a vektorgrafikus kép generálásakor.

# 4.3. Gyenge elemek és kulcsok.

Az 1.2.1. fejezetben, amikor ismertettem az E – K diagram elemeit, említettem a kulcsokat, illetve a gyenge egyedeket. Ezeknek a beállítását a szerkesztőablakon keresztül lehet elvégezni a konverziós folyamat futása alatt. Erre az alkalmazás automatikusan nem képes.

A kulcsok megtalálásához a szöveg sikeres érzékelése szükséges, illetve felismerni, hogy alá van-e húzva az adott rész. Az első nem minden esetben sikerült, a második pedig egyszer sem, ezért a felhasználónak kell beállítani.

A gyenge elemeket vizuálisan a dupla körvonal jelöli. Ennek a tulajdonságnak a megtalálását nem tudtam konzisztensen megoldani, ezért a kulcs tulajdonsághoz hasonlóan a felhasználó tudja beállítani majd a vizuális felületen.

# 4.4. XML fájl készítése

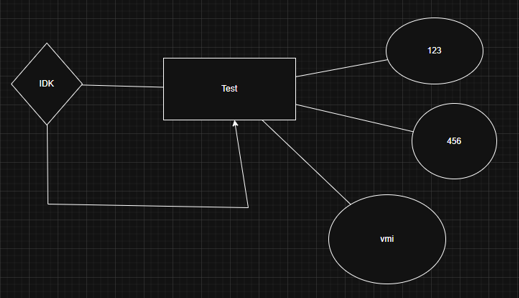
Ha a felhasználó végzett minden szerkesztéssel, az alkalmazás az összes eddig összegyűjtött információ alapján legenerálja a vektorgrafikus képet, amit ezt követően meg lehet nyitni *Drawio* segítségével. Ezek a fájlok lényegükben az XML struktúrát követik, szóval a *Python* beépített xml modulja tökéletes a létrehozásukhoz.

Első lépésként létrehozom az XML fejlécet és a *mxfile* gyökérelemet, amely a fájl megnyitásához és az alkalmazás információk átadásához szükséges. Ezt követően létrehozom a színteret, amiben később elhelyezem a diagrammot, és beállítom a paramétereit, majd elhelyezek két cellát. Ezeknek az a feladata, hogy a többi komponens elhelyezését segítsék, nem jelennek meg magukban.

Második lépésként végig megyek a tömbön, ami tartalmazza az alakzatokat. A kigyűjtött információk alapján mindegyiknek létrehozok egy cellát, és azon belül egy geometriai elemet. Ez megkapja a koordinátát, a szélesség és magasság értékeket, a benne lévő szöveget, valamint a test típusát. A szerkesztőfelületen módosított értékeket is itt dolgozza fel a kód.

A harmadik lépésben a vonalakat dolgozom fel. A testekhez hasonlóan ezek is cellákban helyezkednek el és azon belül egy-egy geometriai elemként vannak definiálva. A fő különbség, hogy nem kell külön beállítani a típust, csak ha nyíl van a vonal végén, illetve rendelkeznek két pontelemmel, amely a kezdő- és végpontot jelenítik meg. Ezen felül beállítom, hogy melyik alakzatokhoz csatlakoznak, azoknak az azonosítóját megadva.

Végül, hogy az így keletkezett elemek ne egy vonalban jelenjenek meg, ha megnyitom szövegszerkesztővel, hierarchia szerint sorokra, illetve indentálási szintekre bontom a nyers kódot. A 4.3. ábrán látható az eredmény, amit a 3.1. ábra feldolgozásával készítettem.

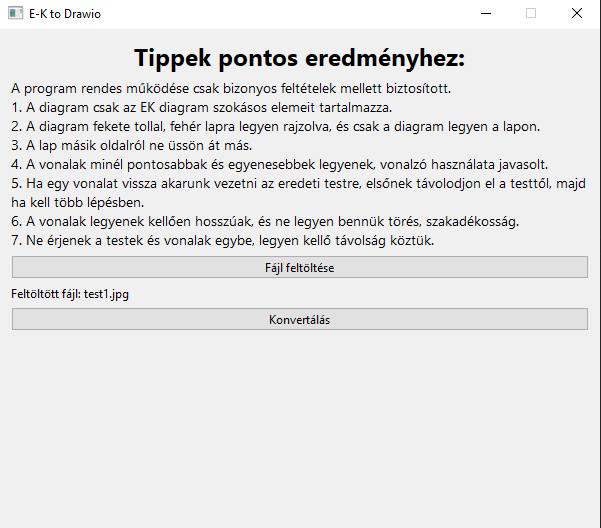


**4.3. ábra – A vektorgrafikus diagram, Drawio-ban megnyitva**

# 5.Alkalmazás használata

Ebben a fejezetben a felhasználói felülethez létrehozott ablakokat és használatukat fogom bemutatni képernyőképekkel kiegészítve, valamint ismertetek néhány érdekesebb hibát, amellyel az alkalmazás megvalósítása közben találkoztam

# 5.1 Főablak használata.

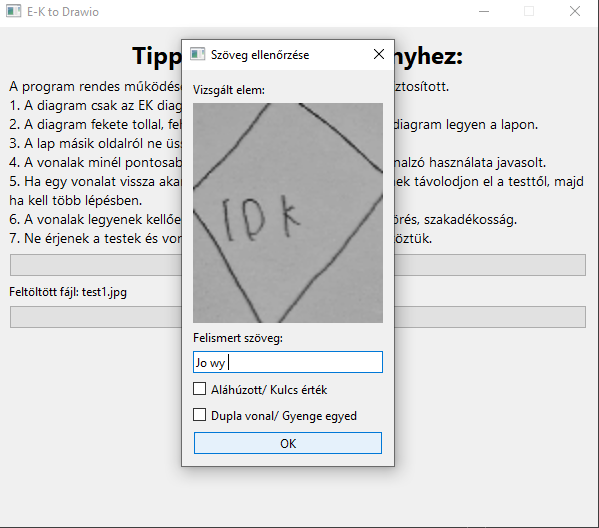


**5.1. ábra – Az alkalmazás főablaka**

Az alkalmazás elindításakor elsőnek a főablak jelenik meg, ezt láthatjuk a 5.1. ábrán. Az ablak felső része tartalmaz néhány tippet, hasznos tanácsot, amelyek segítenek a felhasználónak pontos eredményt elérni. Ez alatt látható a „Fájl feltöltésé” gomb. Ha ezt megnyomjuk, akkor fájlkezelőn keresztül választhatunk egy képet (jpeg, png), amit az alkalmazás lemásol. A kiválasztott fájl-t a gomb alatt láthatjuk.

Ezt követően a „Konvertálás” gomb megnyomására elindul a vektorgrafikus diagram készítő folyamat, amelyet a 3. és 4. fejezetben ismertettem.

# 5.2. Szerkesztőablak használata



**5.2. ábra – A szerkesztőablak**

A konverziós folyamat során, ahogy azt már említettem, (4.2. és 4.3. fejezetek) lehetőségünk van egyes információk szerkesztésére, javítására. A 5.2. ábrán láthatjuk az erre szolgáló felületet. Felül megjelenik az éppen vizsgált elem, hogy tudja a felhasználó, mit tud éppen szerkeszteni. Alatta elsőnek egy szövegmező szerepel, ami tartalmazza a OCR által felismert szöveget. Amennyiben ez nem megfelelő, vagy szerkeszteni kívánjuk, csak át kell írni.

A szöveges mező alatti első jelölőnégyzet arra vonatkozik, hogy az éppen vizsgált elem kulcsot tartalmaz-e vagy nem. Ha be van jelölve, akkor az adott alakzat ezt indikáló adattagja igazra lesz állítva és az éppen megjelenő alakzat szövege a vektorgrafikus diagrammon alá lesz húzva.

A lentebbi checkbox arra vonatkozik, hogy gyenge-e ez a kapcsolat, egyed. Bejelölés esetén, az aláhúzáshoz hasonlóan, az ezt jelölő adattag igaz értéket fog viselni. Eredményül a készített diagramnak ez az eleme duplavonalas körvonalat fog kapni.

Az „OK” gomb megnyomására a program tovább lép a következő elemre. Ha befejeződött a futása, akkor egy felugró ablak kiírja, hogy sikeres volt a folyamat. Az eredményt pedig „*result.drawio*” néven találhatjuk meg az alkalmazás mappájában.

# 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. 26.

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: 2025.05.12.

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 Szelepcsényi Dávid programtervező informatikus BSc szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Intézet Képfeldolgozás és Számítógépes Grafika Tanszékén készítettem, programtervező informatikus BSc 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.