SAPIENTIA ERDÉLYI MAGYAR TUDOMÁNYEGYETEM
MAROSVÁSÁRHELYI KAR,
INFORMATIKA SZAK

SAPIENTIA
ERDÉLYI MAGYAR
TUDOMÁNYEGYETEM

Genetikus algoritmusok alkalmazása a mintfelismerésben

DIPLOMADOLGOZAT

Témavezető:
Horobeț Emil,
Docens

Végzős hallgató:
Szabó Zoltán

2023

UNIVERSITATEA SAPIENTIA DIN CLUJ-NAPOCA
FACULTATEA DE ȘTIINȚE TEHNICE ȘI UMANISTE,
SPECIALIZAREA INFORMATICĂ

UNIVERSITATEA
SAPIENTIA

Folosirea algoritmelor genetice în recunoașterea modelelor

LUCRARE DE DIPLOMĂ

Coordonator științific:
Horobeț Emil,
Conferențiar universitar

Absolvent:
Szabó Zoltán

2023

SAPIENTIA HUNGARIAN UNIVERSITY OF TRANSYLVANIA
FACULTY OF TECHNICAL AND HUMAN SCIENCES
COMPUTER SCIENCE SPECIALIZATION

SAPIENTIA
HUNGARIAN UNIVERSITY
OF TRANSYLVANIA

Genetic algorithms in pattern recognition

BACHELOR THESIS

Scientific advisor:
Horobeț Emil,
Associate professor

Student:
Szabó Zoltán

2023

Declaraţie

Subsemnatul/a .....SZABÓ ZOLTÁN....., absolvent(ă) al/a specializării .....INFORMATICĂ....., promoţia..2022... cunoscând prevederile Legii Educaţiei Naţionale 1/2011 şi a Codului de etică şi deontologie profesională a Universităţii Sapientia cu privire la furt intelectual declar pe propria răspundere că prezenta lucrare de licenţă/proiect de diplomă/disertaţie se bazează pe activitatea personală, cercetarea/proiectarea este efectuată de mine, informaţiile şi datele preluate din literatura de specialitate sunt citate în mod corespunzător.

Localitatea, Tg. Mureş
Data: 2023.06.09

Absolvent

Semnătura.....Szabó.....

Kivonat

Napjainkban, a minta felismerő programok, egyre nagyobb szerepet játszanak. Minta felismerő programokat alkalmazhatunk a hétköznapi életben mint például a snapchat filterek esetében, de alkalmazhatjuk a mezőgazdaságban is, mint például egy drón programja, ami vizsgálja a növények épségét. A minta felismerésben manapság neurális hálókat alkalmazunk, de ezeknek számos hátránya is van. Ezek a problémák közé tartozik a nagy tárhelyhasználat, illetve a tanítatás költségei.

Dolgozatomban bemutatom a genetikus algoritmusok használatát a minta felismerésben. Bemutatom a genetikus algoritmus fogalmát, illetve adok példát ennek működésére. Ezután bemutatom az általam felhasznált genetikus algoritmust, amit az attribútum láncok köré építettem annak érdekében, hogy egy minta felismerő programot hozzak létre.

Dolgozatomban egy bemeneti képen lévő sokszöget összehasonlítok egy bizonyos előre definiált sokszöghalmazzal, és megvizsgálom, hogy melyik sokszög szerepel a bemeneti sokszögben. A keresést genetikus algoritmussal végzem úgy, hogy egy fitnesz értéket több generáción keresztül vizsgálok, majd a legjobb értékű tag lesz a megoldás. Minden generáció tagjait az előző generáció tagjainak keresztezésével, mutációjával és elitizmussal kapom meg.

Rezumat

În zilele noastre, programele de recunoaștere a modelelor joacă un rol din ce în ce mai important. Programele de recunoaștere a modelelor pot fi folosite în viața de zi cu zi, cum ar fi filtrele Snapchat, dar pot fi folosite și în agricultură, cum ar fi un program de drone care examinează starea de sănătate a plantelor. În recunoașterea modelelor sunt folosite reţele neuronale, dar au și câteva dezavantaje. Aceste probleme includ stocarea unei cantităţi mari de date și costurile ridicate de predare, asimilare a informatiilor.

În teza mea, prezint utilizarea algoritmilor genetici în recunoașterea modelelor. Prezint conceptul de algoritm genetic și exemplific funcţionarea acesteia. În continuare, vă prezint algoritmul genetic folosit, pe care l-am construit în jurul lanțurilor de atribute cu scopul de a crea un program de recunoaștere a modelelor.

În această lucrare, compar un poligon dintr-o imagine de intrare cu un anumit set predefinit de poligoane și examinez ce poligoane sunt incluse în poligonul de intrare. Efectuez căutarea cu un algoritm genetic examinând o valoare de fitness pe mai multe generații, iar apoi membrul cu cea mai bună valoare devine soluția. Obțin membrii fiecărei generații prin elitezare, încrucișarea și mutarea membrilor elitezare generației precedente.

Abstract

Nowadays, pattern recognition programs play an increasingly important role. Pattern recognition programs can be used in everyday life, such as in the case of snapchat filters, but can also be used in agriculture, such as a drone program that examines the health of plants. Nowadays, neural networks are used in pattern recognition, but they also have several disadvantages. These problems include high storage usage and teaching costs.

In my thesis, I present the use of genetic algorithms in pattern recognition. I present the concept of the genetic algorithm and give an example of how it works. Then I present the genetic algorithm I use, which I built around attribute chains in order to create a pattern recognition program.

In my thesis, I compare a polygon in an input image with a certain predefined set of polygons and examine which polygon is included in the input polygon. I perform the search with a genetic algorithm, so that I examine a fitness value over several generations, and then the member with the best value becomes the solution. I get the members in each generation by crossing and mutating the members of the previous generation and by using elitism on the previous generation.

Tartalomjegyzék

1. fejezet

Bevezető

Felmerülhet bennünk a kérdés, hogy mik is a genetikus algoritmusok és mire is tudnánk felhasználni őket. A genetikus algoritmusok próbálnak, egy problémát megoldani felhasználva az evolúcióhoz hasonló megoldást, miszerint egy kezdő populációból egy új generációt generálunk azáltal, hogy a legjobb egyedeket keresztezzük, illetve mutációkat idézünk elő bennük. Ezeket a lépéseket ismételjük, amíg egy nekünk megfelelő egyedet, egyedeket kapunk. A genetikus algoritmusok segítséget nyújtanak egy probléma optimális megoldására.

Genetikus algoritmusokat használunk fejlett mérnöki problémák megoldására, mint például egy űrszatellit antennája optimális formájának megtalálása, vagy egy repülőszárny aerodinamikájának legmegfelelőbb alakjának legyártása. Genetikus algoritmusokat használhatunk egyszerűbb problémákban is mint például egy ember hatékony időbeosztása.

Az emberiség sikeréhez nagy mértékben hozzá járult a mintafelislmerés , ugyanis ennek segítségével tudtak az őseink felismerni ragadozókat, illetve étel után kutatni. A minta felismerés egy nagyon kihívó probléma, amivel rengeteg ember foglalkozik, mivel nagyon fontos, hogy elkerüljük a hibákat és minél pontosabb minta felismerő programot hozzunk létre, ugyanis a hibás mintafelismerés nagy veszélyekhez is vezethet. A minta felismerő programok kritikus fontosságúak az iparban, a mezőgazdaságban, a hadseregben, a biztonságban, illetve a szórakoztató iparban. Például egy ipari robotkar felismeri a hibás terméket a csomagolás előtt és eltávolítja ezeket, vagy a hadseregi drón felismeri a tankokat szállító vonatot és semlegesíti ezeket.

A mai világban a mesterséges intelligenciának egyre szélesebb körű használatára kerül sor. Ezek között szerepel a minta felismerés is. Habár a mesterséges intelligencia használata sikeres a minta felismerés terén, mégis a tanítása, az adatainak tárolása nagyon költséges. Erre a problémára szeretnék megoldást nyújtani a projektem által. A projektem egy minta felismerő program ami, a genetikus algoritmusokra épül, felhasználva az attribútum láncokat.

2. fejezet

Elméleti megalapozás

2.1. Genetikus algoritmusok

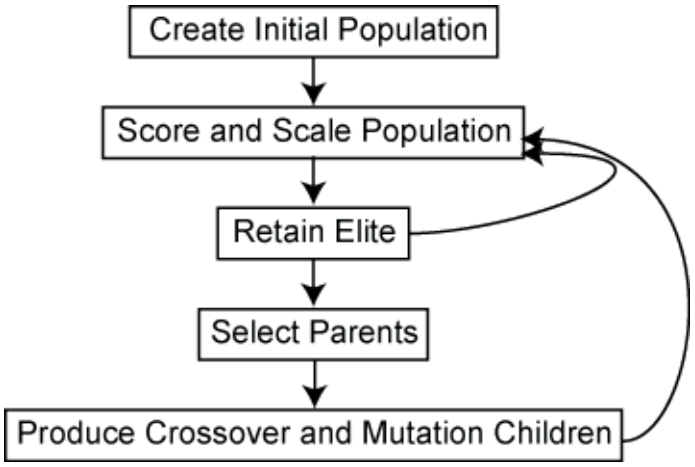
Mit is nevezünk tulajdonképpen genetikus algoritmusoknak? A genetikus algoritmusok a természetben megjelenő biologiai evolucióhoz hasonló megoldást nyújtanak optimalizálási problémák megoldására. A genetikus algoritmusok egy populáció tagjait vizsgálják illetve változtatják a probléma megoldása érdekében.

Minden lépésben a genetikus algoritmus megvizsgálja az egyedeket, és pontozza ezeket, annak függvényében, hogy menyire közelítik meg a megoldást. Ezt a pontozást az úgynevezett fitnesz tartalmazza, amit egy fitnesz számító funkció számol ki. Majd a genetikus algoritmus rangsorolja az egyedeket fitneszi értékük szerint. Új generációt hoz létre, méghozzá úgy, hogy operációkat hajt végre az egyedeken. Ezek az operációk közé tartozik a mutáció, illetve a keresztezés. A keresztezés két egyedet választ ki, illetve ezeknek az egyedeknek a tulajdonságaiból létrehozza az utódot. Az utód tartalmaz tulajdonságokat mindkét szülőtől. A mutáció egy egyednek a tulajdonságait változtatja meg. A mutáció ráta alatt azt értjük, hogy az egyed tulajdonságaiból hányat változtathatunk meg. Egyes genetikus algoritmusok átengedik a legjobb egyedeiket az új generációba. A fentebb említett lépéseket addig hajtják végre, amíg el nem érjük a probléma megoldását, vagy megközelítjük ezt. A genetikus algoritmusokat számos olyan optimalizálási problémák megoldására használhatjuk, amelyek nem megfelelőek a szabványos optimalizálási algoritmusokhoz, beleértve azokat a problémákat is, amelyekben a célfüggvény nem folytonos, nem differenciálható, sztochasztikus, vagy nem lineáris.

2.2. Genetikus algoritmusok müködése

A genetikus algoritmusok a következő formában működnek:

1. Az algoritmus kap egy kezdő populációt vagy ennek hiányában generál egyet.
2. Az algoritmus megállapítja a populáció tagjainak a fitneszét.
3. Kiválasztja a populáció legjobb tagjait és utódokat hoz létre belőlük.
4. A leszármazottak egy részét mutálja, majd a legjobb ősökből és leszármazottakból új populációt hoz létre.
5. Az algoritmus addig hajtja végre ezeket a lépéseket, amíg el nem ér egy kritériumig.



2.1. ábra. GA flowchart

2.3. Hátitáska probléma megoldása genetikus algoritmussal

Elszeretnénk menni túrázni. Erre az útra vihetünk egy hátitáskát, amely nem lehet 2 kg súlyosabb. Az utazásra több tárgy áll rendelkezésre. Melyik tárgyakat tudjuk magunkkal vinni, hogy a táska súlyhatárát a legmegfelelőbben kihasználjuk.

1. Szendvics 200g
2. Konzerv 100g
3. 1l palackozott viz 1kg
4. Zseblámpa 400 g
5. Kamera 500g
6. Iránytü 50g
7. Könyv 250g

Az egyedek a következő képpen vannak feltüntetve:

$$(1, 0, 1, 1, 1, 0, 0) \tag{2.1}$$

Az egyesek azt jelentik, hogy az adott indexű tárgy benne van a táskában, illetve a nullások ezeknek hiányát jelentik. Tehát a fent említett táska tartalma a következő: szendvics, 1l palackozott viz, zseblámpa és kamera. Ha az első lépésben nem kapná meg a megoldást, akkor új egyedeket hozna létre a következő módon:

$$Elsöszülő(1, 0, 1, 1, 1, 0, 0) \tag{2.2}$$

$$Másodikszülő(1, 1, 1, 0, 1, 1, 1) \tag{2.3}$$

$$XOR - - - - - - - \tag{2.4}$$

$$Utód(0, 1, 0, 1, 0, 1, 1) \tag{2.5}$$

Illetve mutálhatja ezeket:

$$Egyed, 0.3mutálodásirátával(1, 0, 1, 1, 1, 0, 0) \tag{2.6}$$

$$Egyed, 0.3mutálodásután(1, 1, 1, 0, 1, 0, 0) \tag{2.7}$$

$$Az1 - esilletvea4 - esindexütulajdonságokváltoztak \tag{2.8}$$

Mivel a mutáció ráta 0.3 és az egyed hossza 7 ezért 2 tulajdonságot mutálunk.Ezt az értéket a következő képpen kaptuk: 7 * 0.3 = 2.1, mivel nem tudunk lebegő pontos mennyiségű tulajdonság megváltoztatására, ezért ennek az eredménynek az egész részét vesszük csak figyelembe.

2.4. Hough-transzformáció problémája

Az általánosított Hough-transzformáció (Generalized Hough Transform GHT) egy közismert technika a minta felismerésben. Ámbár ez a módszer használható részleges minták felismeréséhez, a nagy mennyiségű idő, ami ezeknek a folyamatoknak az elvégzéséhez szükséges, lehetetlen a valósidejű programok számára, amelyek nagy adat mennyiséggel rendelkeznek. Ezeknek az algoritmusoknak a teljesítménye függ a modell alakok adataiknak a formájától, illetve attól, hogy hogyan tudjuk ezeket eltárolni. Ezért a keresendő mintákat, egy láncban tároljuk, melynek elemei egy él-szög páros. Ahhoz, hogy a keresendő mintánál elkerüljük az arányok okozta hibákat az éleket normalizáljuk.

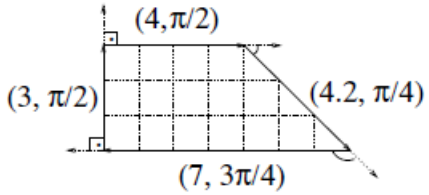
2.5. Atribútum lánc

Amint az előbb említettem, minden egyes sokszög egy attribútum láncból épül fel, ami egy zárt törött vonalból áll, ami mindig visszaérkezik kiindulási pontjába és így sokszöget hoz létre. Egy sokszöget a következőképpen tudunk ábrázolni :

$$(x_1, x_2, ..., x_i, ..., x_n) \tag{2.9}$$

Ahol minden attribútum x_i, két összetevőből áll, méghozzá egy l_i élhosszból és egy hozzátartozó θ_i szögből, amely nem más mint az adott él (l_i) és az előtte levő él (l_{i-1}) által bezárt szög.

$$x_i = (l_i, \theta_i) \tag{2.10}$$



2.2. ábra. Egy sokszög és a hozzátartozó atribútumok

Példa: a következő modellnek az attribútum lánca a következőképpen néz ki:

$$((4, \frac{\pi}{2}), (4.2, \frac{3\pi}{4}), (7, \frac{\pi}{4}), (3, \frac{\pi}{2})) \tag{2.11}$$

Ahhoz, hogy elkerüljük a méret vagy az arány okozta, hibákat az oldal hosszakat normalizáljuk. A normalizálást a következő képpen hajtjuk végre:

$$l'_i = \frac{l_i}{l_{i-1}} \tag{2.12}$$

Tehát a fent megemlített négyszögű alakzatnak, a normalizált attribútum lánca a következő képpen fog kinézni:

$$((1.33, \frac{\pi}{2}), (1.05, \frac{3\pi}{4}), (1.67, \frac{\pi}{4}), (0.43, \frac{\pi}{2})) \tag{2.13}$$

2.6. A genetikus algoritmus bemutatása

A következő részben elmagyarázom a genetikus algoritmus működését.

Adott egy bemeneti kép, aminek n atribútumja van, és ebben a képben S darab modellt keresünk aminek, M tagja van. Az algoritmus részleges minta felismerést alkalmaz, melynek célja, hogy n db attribútumot letérképezzen az M modellekre. A keresési tér hatalmas, méghozzá M^n nagyságú, mivel részleges illesztések is létezhetnek.

Minden egyed letérképez minden bemeneti attribútumot egy megadott modellnek az attribútumára. Minden egyedet egy listával reprezentálunk, ami egy értékpárt tartalmaz, amelynek nem mások mint a modell indexe és az ennek megfelelő attribútuma.

Minden attribútum számításnál normalizáljuk az élhosszakat. Az egyed a bemeneti kép attribútumainak leképzése egy modell attribútumaira.

A bemeneti képet a következő alakban tudjuk felírni:

$$I = (I_1, I_2, ..., I_p, ..., I_n) \tag{2.14}$$

ahol

$$I_p = (l(I_p), \theta(I_p)) \tag{2.15}$$

és

$$|I| = n \tag{2.16}$$

A keresendő alakzatok peddig a következőképpen néznek ki:

$$M = (M_1, M_2, ..., Mj, ..., M_S) \tag{2.17}$$

ahol

$$M_j = (M_{j,1}, M_{j,2}, ..., M_{j,r}, ..., M_{j,m_j}) \tag{2.18}$$

ahol

$$M_{j,r} = (l(M_{j,r}), \theta(M_{j,r})) \tag{2.19}$$

és

$$|M_j| = m_j \tag{2.20}$$

Az egyedeket a következőképpen tudjuk felírni :

$$P = (P_1, P_2, ..., P_k, ..., P_n) \tag{2.21}$$

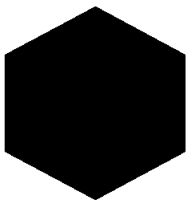
ahol

$$P_k = \nu(I_k) = m_{j,i} \tag{2.22}$$

ahol

$$1 \leq k \leq n \tag{2.23}$$
$$1 \leq j \leq S \tag{2.24}$$
$$1 \leq i \leq m_j \tag{2.25}$$
$$\tag{2.26}$$

Példa: vegyük a következő bemeneti képet



2.3. ábra. Bemeneti kép

és a hozzátartozó keresendő modelleket

2.4. ábra. Modellek

akkor a következő egyedet kapjuk

$$[(0,0),(0,1),(1,1),(0,0),(0,1),(1,3)] \tag{2.27}$$

Az első érték a modell indexét jelképezi, amíg a második a modell attribútumának indexét tartalmazza és ezeknek együtteséből kapjuk meg az egyed elemeit.

Minden egyedet a következőképpen kapunk meg. Először a bemeneti kép k-adik attribútumát (I_k) leképezzük egy véletlenszerű modell (M_j) véletlenszerű attribútumára ($M_{j,r}$). Ezután így folytatódva I_{k+1} , I_{k+2}, ... attribútumokat leképezzük $M_{j,r+1}$, $M_{j,r+2}$, ... attribútumaira, ahol M_j attribútumait körkörösen járjuk be ($M_{j,m}$ után $M_{j,0}$ következik). Miután minden lehetséges attribútumát M_j-nek bejártuk és I-nek még van attribútuma akkor a következő modell alakzatra térünk, vagyis minden I_{k+m} levetítjük egy véletlenszerű $M_{j',r'}$ modell attribútumaira. Tehát folytatjuk a leképezést I_{k+m+1} , I_{k+m+2}, ... attribútumait leképezzük $M_{j',r'+1}$, $M_{j',r'+2}$ ... attribútumaira. Ezt a folyamatot addig ismételjük, amíg ki nem fogyunk I-nek az attribútumaiból.

Az egyedek száma optimalizálás miatt egyenlő a bemeneti kép attribútumai számának a kétszeresével.

Egy egyed, fitnesz értéke, azt jelképezi, hogy mennyire jól illeszkedik minden bemeneti attribútum a keresendő modellek attribútumaira. A fitnesz attól is függ, hogy mennyire konzisztens a modell minden attribútuma a vele szomszédos rá következő attribútummal.

A fitneszt úgy számítjuk ki, hogy teszteljük a bemeneti kép attribútumait, az egyed által leképzett modell attribútumaihoz. A különbséget a bemeneti kép I_k attribútuma és az egyed által letérképezett $\nu(I_k)$ attribútum között egy távolság függvénnyel mérjük. A távolság függvény a következő képpen néz ki:

$$d(I_k,\nu(I_k)) = d_\theta(I_k,\nu(I_k)) + d_l(I_k,\nu(I_k)) \tag{2.28}$$

ahol

$$d_\theta(I_k,\nu(I_k)) = c_\theta abs(\theta(I_k) - \theta(\nu(I_k))) \tag{2.29}$$

Mivel megtörténhet, hogy egy kis számítási hibát kapunk ezért bevezetünk egy hiba tűrő mechanizmust, ami a c_θ konstans lesz és ennek az értéke $\frac{\pi}{18}$.

$$d_l(I_k, \nu(I_k)) = \frac{abs(l(I_k) - l(\nu(I_k)))}{max(l(I_k), l(\nu(I_k)))} \tag{2.30}$$

Két attribútum közötti távolságot egy küszöbértékkel hasonlítjuk össze amely az ϵ aminek értéke 0.01. Ha a távolság kicsi, akkor megegyeznek, máskülönben nem.

$$Matched(I_k, \nu(I_k)) = \begin{cases} 1, & \text{ha } d(I_k, \nu(I_k)) < \epsilon \\ 0, & \text{másküléndben} \end{cases} \tag{2.31}$$

A fitnesz funkció bünteti a modellek számát, amelyekre az egyed nem tudja leképezni a bemeneti attribútumokat. Fitneszt a következő képlettel számítunk:

$$Fitnessz = -(részlegesenfelismertmodellek + anemtaláltbemenetiatributokszáma) \tag{2.32}$$

A genetikus algoritmus lineáris rangsorolási stratégiát alkalmaz. A legjobb egyedek egy harmada továbbjut a következő populációba. Az új egyedeket operátorok segítségével hozza létre.

Kereszteződés (1PTX) által arra gondolunk, hogy kiválasztunk két szülőt majd ezeknek attribútumait úgy adjuk át az utódnak, hogy kiválasztunk egy indexet ami előtt az első szülő attribútumait örökli, míg az adott indextől a második szülő attribútumait örökli.

Mutációnál kiválasztunk véletlenszerűen attribútumokat, amelyekre véletlenszerűen leképezzük a bemeneti kép egy-egy attribútumát egy véletlenszerű modell attribútumára. A mutáció ráta alatt azt értjük, hogy a populáció hány tagját változtathatjuk meg.

3. fejezet

Programok, technológiák bemutatása

3.1. Programozási nyelv (C++)

Programomat a C++ nyelvben írtam. A legnyomosabb oka ennek a döntésnek az, hogy ebben a nyelvben érzem legotthonosabban magam. A C++ nyelv tökéletes a (procedurális) funkció központi programozásra. Programom során nem használtam osztályokat. A program nagyrészt az általam létrehozott struktúrák köré épül. A C++ nyelvnek rengeteg külső könyvtára van, illetve rengeteg fórum van az interneten amely segíteni tud az ebben irt nyelvben szereplő problémákra. A programomban használtam egy nagy külső csomagot, ami nem más mint az OpenCV. Az OpenCV egy vizualitásokkal foglalkozó csomag.

3.2. Visual Studio

A programomat visual studioban írtam, mivel számos előnye van más programozási környezetekhez képest. Első sorban rengeteg tapasztalatom van a használatában. Ingyenes, illetve rengeteg könyvtára van, amit fel tudok használni. Kezelése viszonylag könnyű. Rengeteg kis funciója van, ami segít a kódolásban mint például az IntelliSense, tartalmaz kód faktorizációt. Az integrált hibakereső, forrásszintű hibakeresőként és gépszintű hibakeresőként is működik. Elfogad rengeteg modult, amelyek kibővítik szinte bármely funkcionalitását, beleértve a forrásvezérlő rendszerek támogatását is, mint például a Subversiont és a Gittet, illetve új eszközkészletek hozzáadását, mint például szerkeztök és vizuális tervezőket a tartományspecifikus nyelvekhez, vagy eszközkészleteket a szoftverfejlesztés életciklusaihoz.

3.3. OpenCV

Az OpenCV egy nyílt forráskódú könyvtár, amely több száz vizualitással foglalkozó algoritmust tartalmaz. Vizualitás alatt azt értem, hogy a függvények és tárolók nagy része képfeldolgozással, illetve videó feldolgozással foglalkozik. Az OpenCV-nek rengeteg alkalmazása lehet képfeldolgozáson kivül mint például 2D és 3D funkció eszközkészletek létrehozása, arcfelismerés, videó nyomonkövetés, mozdulatok felismerése, stb.

Az OpneCV algoritmusok és tárolók a cv:: névtér használatával elérhetőek. Az OpenCV támogatja a C,C++ illetve a Python nyelveket és viszonylag nem nehéz a használata.

Az OpenCV számos alkönyvtárának viszonylag kis részét használtam. Az általam felhasznált könyvtárak a képfeldolgozással foglalkoznak és ezeknek a könyvtáraknak a felhasznált elemeik a következők:

1. cv::Mat tömbhöz hasonlitó tároló
2. cv::Point pontok tárolására szolgálo tároló
3. cv::imread() képek beolvasására alkalmas függvény, egy fájlból kiveszi a képet és ezt eltárolja
4. cv::imshow() a képet megjeleníti egy ablakban
5. cv::waitKey() a függvény várakozik egy karakterre
6. cv::cvtColor() a kép színtartományát módosítja
7. blur() elhomályosítja a képet
8. cv::Canny() Canny szűrőt alkalmaz a képre
9. cv::findContours() megkeresei a kép összes kontürjét
10. cv::approxPolyDP() a kontüröket végig járja és alakzatokat hoz létre belőlük

4. fejezet

Szoftverspecifikáció

Ebben a fejezetben bemutatom a program működéséhez szükséges dolgokat.

4.1. Felhasználói követelmények

1. Kell a Visual Studio.
2. Szükséges az OpenCV , instalálása, illetve csatlakoztatása az IDE-hez.
3. A felhasználó képes kell legyen új modell minták,illetve bemeneti képek megadására, ehhez szükséges egy képlétrehozó szoftware mint például a paint.

4.2. Rendszer követelmények

1. a bementi kép ideiglenes tárolásának biztosítása.
2. a bemeneti atribútumláncok tárolása, illetve felismerése.
3. új generációk térehozása és vizsgálata.
4. az egyedek vizsgálása és a rajtuk elvégezhetö müveletek elvégzése.
5. az eredmények eltárolása, illetve vizsgálata.

5. fejezet

Szoftver

A következö fejezetben a program bemutatására kerül sor. A programot amint említettem a visual studioban C++ nyelvben írtam meg. A programm elején létrehoztam az általam irt struktokat, amelyek a program központjában álnak.

```
struct Atributum_list {
    // oldal,szog
    std::vector<std::pair<double,double>> featuresA;
};
struct Individual {
    //Model_index , Model_atributuma
    std::vector<std::pair<int, int>> featuresI;
    double fitness;
    std::vector<int> models;
};
```

5.1. ábra. Struktok bemutatása

Az Atributum_list az atribútum láncokat tároló struktura, amely egy olyan vektorból áll, amelynek elemei egy érték páros. Az első érték egy élhosszat tárol, míg a második érték egy szöget tárol, amely nem más mint az oldal és az azelötti oldal által bezárt szög. Ezt a struktot használom a bemeneti sokszög, illetve a keresendő modellek atribútumjainak tárolására.

Az Individual struktot az egyedek adatainak tárolására használjuk. Három tagja van ennek a struktnak. Az első tag egy vektorból áll, amelynek első eleme a model indexét tartalmazza, a második érték peddig ezen atribútum indexét tartalmazza. Annak érdekében hogy megkapjuk az egyed atribútumait, ezt vissza kell fejtsük a következő képpen. A keresendő modellek listájában megnézzük a vektor első értékét és a vele azonos indexü modellnek megnézzük a második érték indexén lévő atribútumot. A struct második értéke fogja tárolni az egyed fitnesszi értékét. Ez az érték adja meg, hogy menyire jól tudnak illeszkedni a bemenő sokszög atribútumai a keresendö modellek atribútumaira. A struct harmadik tagja peddig tartalmazza a benne parciálisan vagy egészben felismert modellek indexét.

A program elején beolvassuk a bemeneti képet, illetve a keresendő modelleket. Ezért az OpenCv imread() függvény felelős, illetve ezt egy Mat változóban fogjuk tárolni későbbi feldolgozásra.

```
cv::Mat src1 = cv::imread("images\\model3.png"); //shapeM mit, keresendo modellek
if (!src1.data) { std::cout << "Src1" << std::endl; return 0; }
cv::Mat src2 = cv::imread("images\\input4.png");//shapeI miben, bemeneti kep
if (!src2.data) { std::cout << "Src2" << std::endl; return 0; }
```

5.2. ábra. Beolvasás

Annak érdekében hogy könnyebben feltudjuk ismerni a modellek köralakjának a képekre egy Canny szűrőt alkalmazunk. Ennek érdekében a képeket szürke árnyalatba teszük, majd elhomájosítjuk ezeket. A képek szürkeárnyalatáért felelős funkció az OpenCv cvtColor(). A homályosítás az OpenCv blur() funkciójának használatával történik. A Canny szőrésért szintén az OpenCv egy funkciója felelős, aminek a neve megegyezik a szőrő nevével, ami nem más mint Canny().

```
cv::cvtColor(src1, gray1, cv::COLOR_BGR2GRAY);
blur(gray1, bw1, cv::Size(3, 3));
cv::Canny(gray1, bw1, 80, 240, 3);
cv::cvtColor(src2, gray2, cv::COLOR_BGR2GRAY);
blur(gray2, bw2, cv::Size(3, 3));
cv::Canny(gray2, bw2, 80, 240, 3);


cv::findContours(bw1.clone(), contours1, cv::RETR_EXTERNAL, cv::CHAIN_APPROX_SIMPLE);
cv::findContours(bw2.clone(), contours2, cv::RETR_EXTERNAL, cv::CHAIN_APPROX_SIMPLE);
```

5.3. ábra. Canny

A Canny szőrő elvégzése után a bemeneti kép, illetve a keresendő modelleknek a körvonalat a contours vektorokba helyezzük. Ezt az OpenCv findContours() funkciója végzi. Mivel a findContours minden kontürt tartalmaz, a bemenő adatokra a approxPolyDP OpenCv függvényt hívom meg, ami különszedi a bemenő képen szereplő alakzatokat.

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

    cv::approxPolyDP(cv::Mat(contours[i]), approx, cv::arcLength(cv::Mat(contours[i]), true) * 0.0015, true);
    Atributum_list als = (fit_into_shape(approx));
    shapeM.push_back(als);
}
/*
int db=0;
for (int i = 0; i < shapeM.size(); i++)
{
    db+=shapeM[i].featuresA.size();
    print_A(shapeM[i]);
    std::cout << "shapeM[" << i << "].size=" << shapeM[i].featuresA.size() << std::endl;
}
std::cout<<"shapeM osszes atributuma: "<<db<<std::endl;
*/
cv::findContours(bw2.clone(), contours, cv::RETR_EXTERNAL, cv::CHAIN_APPROX_SIMPLE);
//print_A(shapeI);
for (int i = 0; i < contours.size(); i++)
{
    cv::approxPolyDP(cv::Mat(contours[i]), approx, cv::arcLength(cv::Mat(contours[i]), true)* 0.0015, true);
    Atributum_list als = (fit_into_shape(approx));
    shapeI=als;
}
std::cout << "shapeI.size()=" << shapeI.featuresA.size() << std::endl;
```

5.4. ábra. Contours

Mivel az OpenCv a Mat illetve más egyedi tárolokban tárolja az adatokat, szükséges volt egy átalakító funkciót írnom. Ez a funkció a fit_into_shape. A fit_into_shape egy Point-okból állo vektort dolgoz fel és ezekből egy Atributum_list-et térit vissza(Atribútum láncot). A fit_into_shape három más segéd funkciót használ, amik nem mások mint a distance, az angle és a normalized. Az angle az atributum lánc szögeit számítja ki radiántban, míg a distance az élhosszakat adja meg és végül a normalized normalizálja az egész atribútum láncot.

```
Atributum_list fit_into_shape(std::vector<cv::Point> point_l)
{
    Atributum_list shape;
    int i;
    shape.featuresA.push_back(std::make_pair(distance(point_l[0], point_l[1]), angle(point_l[point_l.size()], point_l[0], point_l[1])));
    for (i = 1; i < point_l.size()-1; i++)
    {
        shape.featuresA.push_back(std::make_pair(distance(point_l[i],point_l[i+1]), angle(point_l[i-1],point_l[i],point_l[i+1])));

    }
    shape.featuresA.push_back(std::make_pair(distance(point_l[point_l.size()], point_l[0]), angle(point_l[point_l.size() - 1], point_l[point_l.size()], point_l[0])));

    return normalized(shape);
}
double distance_between_edges(double e1, double e2)
```

5.5. ábra. fit_into_shape

```
static double angle(cv::Point pt1, cv::Point pt2, cv::Point pt0)
{
    double dx1 = pt1.x - pt0.x;
    double dy1 = pt1.y - pt0.y;
    double dx2 = pt2.x - pt0.x;
    double dy2 = pt2.y - pt0.y;
    return (dx1 * dx2 + dy1 * dy2) / sqrt((dx1 * dx1 + dy1 * dy1) * (dx2 * dx2 + dy2 * dy2) + 1e-10);
}
double distance(cv::Point p1, cv::Point p2)
{
    return cv::norm(p1 - p2);
}
Atributum_list normalized(Atributum_list shape)
{
    Atributum_list shape_m;
    shape_m.featuresA.push_back(std::make_pair(shape.featuresA[0].first/shape.featuresA[shape.featuresA.size()].first,shape.featuresA[0].second));
    for (int i = 1; i < shape.featuresA.size(); i++)
    {
        shape_m.featuresA.push_back(std::make_pair(shape.featuresA[i].first / shape.featuresA[i-1].first, shape.featuresA[i].second));
    }
    return shape_m;
}
```

5.6. ábra. fit_into_shape segéd fügvényei

A bemeneti kép atribútumláncát a shapeI-ben fogjuk tárolni, míg a keresendő modellek atribútumláncait a shapeM fogja ezentul tárolni. Mivel meg van, hogy miben mit kell keresni ezért ezután a genetikus algoritmus következik. Legelőször a populációt fogjuk inicializálni, ezért a population_init funkció felelős. A populáció tagjai száma a bemeneti kép atribútumai számának kétszeresével egyenlő. Minden egyed atribútum száma megegyezik a bemeneti kép atribútumainak számával. Az egyedek, amint a program elején feltűntettem szintén atribútumokból állnak. Az egyedek viszont nem egy atribútum láncból vannak felépitve, a hanem keresendő modellek atribútumaiból. Ezért a populáció tagjait véletlenszerű modellek atribútumaiból épitjük fel.

```
std::vector<Individual> population_init(std::vector<Atributum_list> shapeasM, Atributum_list shapeI)
{
    std::vector<Individual> population;
    while (population.size() < shapeI.featuresA.size() * 2)
    {
        Individual individual;
        for (int i = 0; i < shapeI.featuresA.size(); i++)
        {
            int k = rand() % shapeasM.size();
            int p = rand() % shapeasM[k].featuresA.size();
            individual.featuresI.push_back(std::make_pair(k, p));
        }
        calculate_fitness(individual,shapeasM,shapeI);
        population.push_back(individual);

    }
    return population;
}
```

5.7. ábra. populáció inicializálása

A populáció inincializálása után már csak az van hátra, hogy új generációkat hozzunk létre és ezeknek a végén megviszgáljuk a legjobb értékű egyedet. A programomban a generációk számát 50-re állítottam, de ez az érték szabadon változtatható. Az új generációk létrehozásáért a new_population() függvény felelős, amely a régi populáció tagjainak a legjobb egyedeiből, ezeknek leszármazotaiból, illetve ezekek mutálásából áll.

```
std::vector<Individual> pop;
pop=population_init(shapeM, shapeI);
for (int g = 0; g < 50; g++)
{
    std::cout << "gen= " << g << std::endl;
    pop = new_population(pop, shapeM, shapeI);
    sort_population(pop);
    for (int i = 0; i < pop.size(); i++)
    {
        //print_I(pop[i]);
        //print_Iv(pop[i],shapeM);
        //std::cout <<i<<"-nek fitnesze= " << pop[i].fitness << std::endl;
        /*
        std::cout << i << "-nek modeljei: " ;
        for (int i = 0; i < pop[0].models.size(); i++)
        {
            std::cout << pop[0].models[i] << " ";
        }
        std::cout << std::endl;*/

    }
    std::cout << "A generacio legjobb tahjanak fitnessze = "<<pop[0].fitness<<std::endl;
    std::cout << "A generacio atlagos fitnessze = " << avg_fitnnes(pop) << std::endl;
}
```

5.8. ábra. generációk

```
std::vector<Individual> new_population(std::vector<Individual> population, std::vector<Atributum_list> shapesM, Atributum_l
{
    sort_population(population);
    std::vector<Individual> new_pop;
    Individual individual;
    int i = 0, j = 0, x, y, m = 0, e;
    for (i = 0; i < population.size() / 3; i++)
    {
        new_pop.push_back(population[i]);
    }
    while (new_pop.size() < population.size())
    {
        x = rand() % population.size()*2/3;
        y = rand() % population.size()*2/3;
        e = rand() % 2;
        if (m < population.size() / 10 && e)
        {
            new_pop.push_back(mutate(crossover(population[x], population[y]), shapesM));
            m++;
            continue;
        }
        new_pop.push_back(crossover(population[x], population[y]));
    }
    for (int i = 0; i < new_pop.size(); i++)
    {
        calculate_fitness(new_pop[i], shapesM, shapeI);
    }

    return new_pop;
}
```

5.9. ábra. new_population

Amint a fenti ábrán láthatjuk a populáció tagjait sorrendbe tudjuk tenni. A sorrend a fitnesz érték alapján történik. A legnagyobb fitnesz értékkel rendelkező egyedek jobbnak minősülnek mint az alacsony fitnesz értékű tagok. A rangsorolás a következő képpen történik.

```
bool compare_fitness(Individual s1, Individual s2)
{
    return s1.fitness > s2.fitness;
}
void sort_population(std::vector<Individual> &population)
{
    std::sort(population.begin(), population.end(), compare_fitness);
}
```

5.10. ábra. new_population

Az új populációnak az egyedeit három módon kaphatjuk meg. Az első ilyen mód az elitizmus. Elitizmus alatt azt értjük, hogy az előbbi populáció tagjainak legjobb egyharmada tovább jut az új populációba. A második metodus a keresztezés. Keresztezésnél kiválasztjuk véletlenszerűen az előbbi populáció két egyedét és ezeknek atribútumait keverjük. A harmadik módszer a mutáció.

```
Individual crossover(Individual s1, Individual s2)
{
    Individual shape;
    int k, i, j;
    k = rand() % s1.featuresI.size();
    for (i = 0; i < k; i++)
    {
        shape.featuresI.push_back(s1.featuresI[i]);
    }
    for (j = k; j < s2.featuresI.size(); j++)
    {
        shape.featuresI.push_back(s2.featuresI[j]);
    }

    return shape;
}
```

5.11. ábra. keresztezés(1PTX)

A keresztezés a következő képpen zajlik. Kiválasztunk egy véletlenszerű számot 0 és a bemeneti kép atribútumai száma között. Ez a szám egy indexet fogg jelölni. Az új egyed az első szülő atribútumait fogja örökölni az adott indexig. Azután az utód a második szülő atribútumait fogja örökölni. Az utód atribútumainak a száma megegyezik a szülők atribútumainak a számával. Ezekért a lépésekért felelős függvény a crossover().

```
Individual mutate(Individual individual, std::vector<Atributum_list> model_shapes)
{
    Individual shapes_m;
    int r, m, j;
    r = rand() % individual.featuresI.size();
    for (int i = 0; i < individual.featuresI.size(); i++)
    {

        if (i == r)
        {
            m = rand() % model_shapes.size();
            j = rand() % model_shapes[m].featuresA.size();
            shapes_m.featuresI.push_back(std::make_pair(m, j));

        }
        else
        {
            shapes_m.featuresI.push_back(individual.featuresI[i]);
        }
    }
    return shapes_m;
}
```

5.12. ábra. mutáció

A mutáció egy ujonnan létrehozott egyednek egy atribútumát kicseréli egy véletlenszerűen kiválasztott modell atributumával. A mutáció ráta azt jelenti, hogy a populáció hány tagján végezzük el ezt a mutációt. A programom mutáció rátája az egyedek egyharmada. A mutációért felelős függvény a mutate()

```
void calculate_fitness(Individual& individual, std::vector<Atributum_list> shapesM, Atributum_list shapesI)
{
    Individual temp;
    std::vector<std::vector<int>> model;

    for (int i = 0; i < shapesI.featuresA.size(); i++)
    {
        for (int j = 0; j < individual.featuresI.size(); j++)
        {
            if (matches(shapesI.featuresA[i], shapesM[individual.featuresI[j].first].featuresA[individual.featuresI[j].second]))
            {
                temp.featuresI.push_back(individual.featuresI[i]);
            }
        }
    }
    check_model(temp,shapesM);
    individual.fitness = temp.fitness;
    individual.models = temp.models;

}
```

5.13. ábra. calculat_fitness

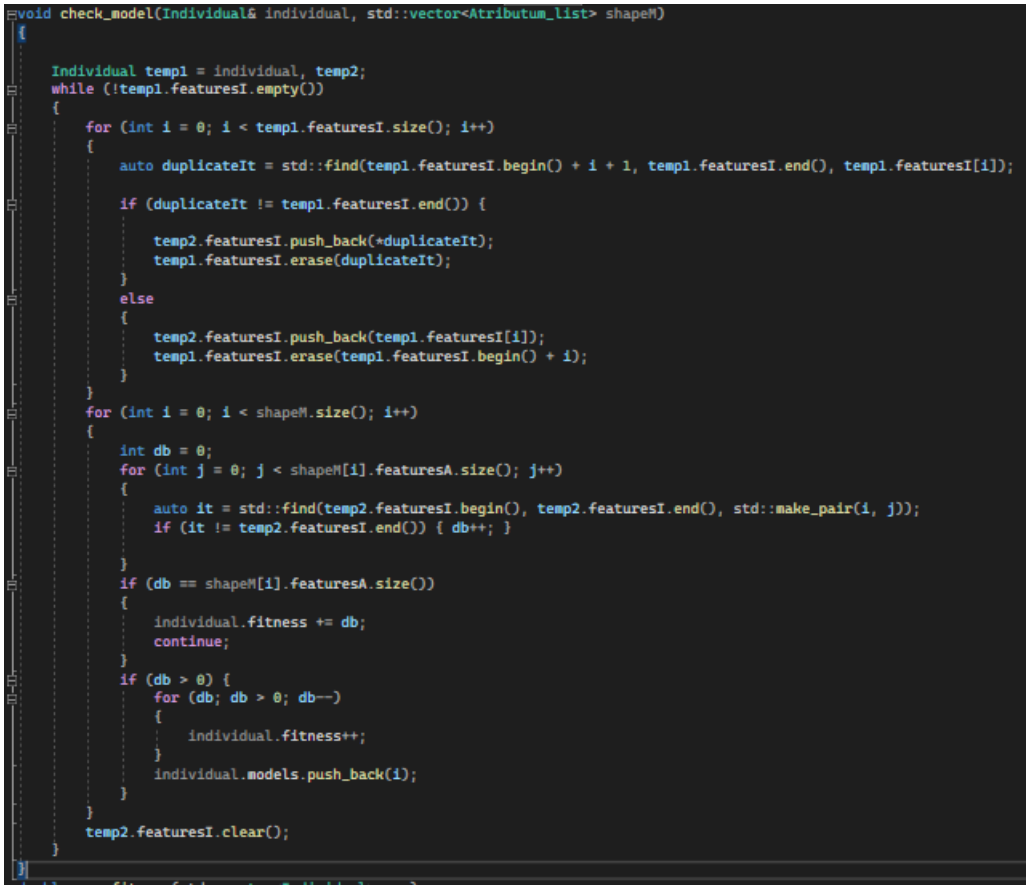
Az egyedek fitnessz értékéért a calculate_fitness() felelős. A fügvény ellenörzi, hogy a bemeneti kép atribútumai és az egyed által reprezentált modell alakok atribútumai megegyeznek-e. Ha igen, akkor ezeket elmenti egy új egyedben, amit további feldolgozásra küld, majd megadja, hogy melyik modellnek atribútumait fedezte fel és az egyed models vektorába beteszi a modell indexét.A függvénynek két segéd függvénye van. Ezek a check_model() és matches().

```
/*
double theta = 0.175;
double epsilon = 0.01;
*/
double distance_between_edges(double e1, double e2)
{
    if (e2 < e1) { return std::fabs(e1 - e2) / e1; }
    return std::fabs(e1 - e2) / e2;
}
double distance_between_angles(double a1, double a2)
{
    return std::fabs(a1 - a2) * theta;
}
bool matches(std::pair<double, double> a1, std::pair<double, double> a2)
{
    double de, da;
    de = distance_between_edges(a1.first, a2.first);
    da = distance_between_angles(a1.second, a2.second);
    if (de - da < epsilon)
    {
        return 1;
    }
    return 0;

}
```

5.14. ábra. matches+distance_between_edges+distance_between_angles

A matches() függvény felelős azért, hogy ellenőrízze a bemeneti modell egy atribútuma megegyezik-e (vagy nagyon hasonlit) egy keresendö modell atribútumával. A hasonlóságot két segéd fügvényel méri, amik nem mások mint distance_between_edges() és distance_between_angles().



```
void check_model(Individual& individual, std::vector<Atributum_list> shapeM)
{

    Individual temp1 = individual, temp2;
    while (!temp1.featuresI.empty())
    {
        for (int i = 0; i < temp1.featuresI.size(); i++)
        {
            auto duplicateIt = std::find(temp1.featuresI.begin() + i + 1, temp1.featuresI.end(), temp1.featuresI[i]);

            if (duplicateIt != temp1.featuresI.end()) {

                temp2.featuresI.push_back(*duplicateIt);
                temp1.featuresI.erase(duplicateIt);
            }
            else
            {
                temp2.featuresI.push_back(temp1.featuresI[i]);
                temp1.featuresI.erase(temp1.featuresI.begin() + i);
            }
        }
        for (int i = 0; i < shapeM.size(); i++)
        {
            int db = 0;
            for (int j = 0; j < shapeM[i].featuresA.size(); j++)
            {
                auto it = std::find(temp2.featuresI.begin(), temp2.featuresI.end(), std::make_pair(i, j));
                if (it != temp2.featuresI.end()) { db++; }

            }
            if (db == shapeM[i].featuresA.size())
            {
                individual.fitness += db;
                continue;
            }
            if (db > 0) {
                for (db; db > 0; db--)
                {
                    individual.fitness++;
                }
                individual.models.push_back(i);
            }
        }
        temp2.featuresI.clear();
    }
}
```

5.15. ábra. check_model

A check_model() függvény feldolgoza a sikeresen felismert atribútumait az egyednek, azáltal, hogy vizsgálja a teljesen vagy csak részlegesen felismert modelleket, illetve ezek elöfordulásának a számát.

Ezek mellett írtam pár kiírással foglalkozó függvényt és egy populáció átlagszámitó függvényt. Ezek a print_A() , a print_I() illetve a print_Iv(). A print_A() kiírja a modellt alkotó élhosszakat és a hozzájuk tartozó szögeket radiántban. A print_I() kiírja az egyednek elemeit egy model és annak indexe alapján. A print_Iv() kiírja az egyed elemeit egy él-szög formában. A populáció átlag fitnesszét számoló függvény a avg_fitnes().

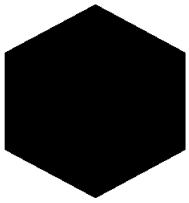
```cpp
double avg_fitnnes(std::vector<Individual> pop)
{
    double fitness = 0;
    for (int i = 0; i < pop.size(); i++)
    {
        fitness += pop[i].fitness;
    }
    return fitness / pop.size();
}
void print_A(Atributum_list alist)
{
    for (int i = 0; i < alist.featuresA.size(); i++)
    {
        std::cout << "Az " << i << "-edik tag = (" << alist.featuresA[i].first <<", " << alist.featuresA[i].second << " )" << std::endl;
    }
}
void print_I(Individual individual)
{
    for (int i = 0; i < individual.featuresI.size(); i++)
    {
        std::cout << "Az " << i << "-edik tag = (" << individual.featuresI[i].first << ", " << individual.featuresI[i].second << " )" << std::endl;
    }
}
void print_Iv(Individual individual, std::vector<Atributum_list> shapeM)
{
    for (int i=0;i<shapeM.size();i++)
    {
        std::cout << "Az " << i << "-edik tag = (" <<
            shapeM[individual.featuresI[i].first].featuresA[individual.featuresI[i].second].first <<
            ", " << shapeM[individual.featuresI[i].first].featuresA[individual.featuresI[i].second].second <<
            " )" << std::endl;
    }
}
```

5.16. ábra. kiíratás+átlagfitnessz

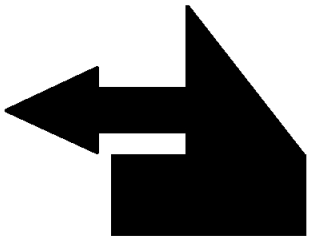
6. fejezet

Kísérlet,eredmények

A kísérletem arról szól, hogy négy különböző bemeneti képen keresek három különdbözó modell halmazt. A bemeneti képek alakzatai, illteve a keresendő modellek alakzatai zárt sokszögek. Ezeket a sokszögeket már normalizált alakba hoztam. A bemeneti képek modelleit a hozzájuk tartozó keresendő modellek fedésével, illetve egymáshoz való illesztésével hoztam létre. A bemeneti képek a következök:

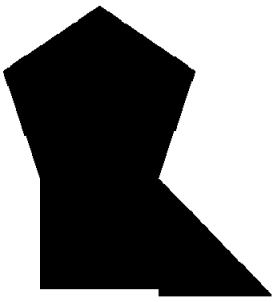


6.1. ábra. Első bemenet



6.2. ábra. Máspdik bemenet

6.3. ábra. Harmadik bemenet

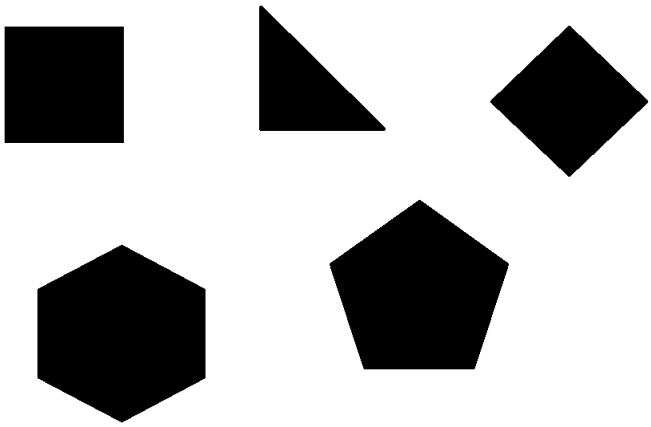


6.4. ábra. Negyedik bemenet

A keresendő modell halmazok a következők:

6.5. ábra. Első modellhalmaz

6.6. ábra. Második modellhalmaz



6.7. ábra. Harmadik modellhalmaz

Ezentúl, a bemeneti képhalmaznál, használni fogom a kicsi i betűt és egy hozzárendelt indexet, ami az adott bemeneteli képnek(modellnek) indexét fogja jelképezni, míg a keresendő modellekre egy kicsi m betűt fogom alkalmazni. A modelleket körkörösen fogom bejárni, azaz az utolsó tag után a legelső jön. Több program futtatást vizsgáltam minden egyes bemeneti képre és a hozzájuk tartozó keresendő modell halmazokra. A vizsgálatok a következők:

Bemeneti alakzat	Atribútumok száma	Keresendő alakzatok(modellek)
i_0	6	i_0
i_0	6	m_0
i_1	13	m_1
i_2	15	m_1
i_3	12	m_2

6.1. táblázat. 0

Számos kísérlet arra szolgált, hogy vizsgáljuk az optimalitást. Egyik kísérlet az i_1 bemeneti kép modelljének vizsgálatát mérte, különböző mutáció arányokkal. Ennél a kísérletnél nem használtam mutációt, illetve kihagytam a keresztezést. A kísérlet eredménye az volt, hogy az algoritmus nem működik optimálisan a mutáció, vagy keresztezés hiányában. Tehát mindkét folyamat szükséges a hatékony működéshez. A különbséget a következő táblázat fogja megmutatni:

Műveletek	Találatok gyakorisága
Mut(0.3), 1PTX	0.32
Mut(0.3)	0.00
1PTX	0.00

6.2. táblázat. 1

A táblázatban feltűntetett adatok, a következö módon voltak megadva:

1. a bemeneti minta i_3 volt.
2. 25-ször hajtottuk végre a kísérletet.
3. A generációk számát korlátoztuk 150-re.
4. A $Mut(0.3)$ a mutáciot jelenti, az egyedeket 30% mutáljuk.
5. $1PTX$ keresztezést jelent.

A kísérleteket a következő táblázatban vannak összegezve:

Be.	Mod.	Fit.	Fit. sz.	Fit.Gen.	Gen. sz.
i_0	i_0	6	0	5.9	0.13
i_0	m_0	2	0	2	0
i_1	m_1	8	0.63	7.89	7.89
i_2	m_1	14	0.47	14.2	0.42
i_3	m_2	5	1.20	5.44	1.20

6.3. táblázat. 2

A táblázatban feltűntetett adatok, a következö módon voltak megadva:

1. Be. bemeneti kép.
2. Mod. Modell halmaz.
3. Fit. a fitnessz értékek átlaga.
4. Fit. sz. a fitnessz értékek szorása.
5. Fit. Gen. az utolso populációinak fitnesszének átlaga.
6. Gen. sz. az utolso populációinak fitnesszének átlagaik szorása

Mivel az algoritmusunk, nem vesz figyelembe méret, ezért felismeri azoknak a keresendő modellek attribútumait is a bemeneti képben, ha azok lekicsinyítve vagy megnövelve vannak. A program figyelemben veszi azokat az attribútumokat is, amelyek elvannak fordítva, amíg a szög és él megmarad.

7. fejezet

Kónkluzió

7.1. Kónkluzió

Az új mintafelismerési modszerek fejlesztésében, használhatjuk a genetikus algoritmusok összekeverését az atributum láncokkal és egy eléggé hatékony programot kapunk belölük. Az alakok körvonalait, egy megtört egyenesként vesszük figyelembe, amit egy atributum láncba teszünk. Minden megtört egyenest, egy él-szög párosban tároljuk el. A normalizálás használata által el tudjuk kerülni a méret arányok miatti felbukanó hibákat.

A kísérleteket nézve, egy eléggé sikeres programot lehet készíteni evel a módszerrel. Ennek a megközelitésnek, számos elönye van. Első sorban sokkal számításbarátibb, mint a kimerítő keresési algoritmusok. Másodsorban, sokkal tárhelybarátságosabb mint egy neurális háloval rendelkezö algortitmus, föleg memória szempontjából. Az algoritmus relativan eléggé gyors, és egy relativan eléggé kis mennyiségű elemet vizsgál a keresési térből. A populációk használata, illetve a rajtuk végzett műveletek kikerülik a fő problémát, amiben a zsugori algoritmusok (greedy alghorithms) szenvednek, ami nem más mint a helyzeti optimális megoldások.Az hátránya az,hogy csak a konvex sokszögekre müködik.

Összefoglaló

Dolgozatomban a genetikus algoritmusokkal foglalkoztam. Először ismertettem az általam használt genetikus algoritmusnak architektúráját majd elmerültem ennek működésében.A továbbiakban ismertettem a technológiákat amin keresztül létre hoztam programom, illetve érveltem, hogy miért választottam a használt környezetet, kiterjesztéseket illetve könyvtárakat. Ezután bemutattam a programomat és ennek működését vizsgáltam egy általam létrehozott kísérleti állománnyal.

Jövőbeli terveimet illetően szeretnék egy megoldást kapni annak érdekében, hogy a bemeneti kép modelljében tudjak modell attribútumokat vizsgálni és így pontosabb megoldást tudjak megadni. Ezen kivül szeretném implementálni a hill climbing módszert, mivel nem volt elég időm rá, illetve egyéb optimalizálási függvények hozzáadását. Legvégül a programnak szeretnék adni egy felhasználói felületett, annak érdekében, hogy a programot könnyedén lehessen felhasználni önmagában is, bármilyen programozási nyelv ismerete nélkül.

Köszönetnyilvánítás

Szeretném megköszönni mindazok munkáját, akik segítettek abban, hogy az általam elkészített projekt és dokumentum létrejöhessen. Köszönöm azoknak a személyeknek, akik útirányt adtak a projekt felépítésében és legvégül, de nem utolsó sorban téma vezetőtanáromnak.

Külön köszönetet szeretnék nyilvánítani, azoknak az embereknek, akiknek a könyveiből tanulmányozhattam a genetikus algoritmusok működését illetve ezeknek használatát a minta felismerésben, amivel hozzásegítettek felépíteni a saját programom változatát.

A legnagyobb segítségemre az internet szolgált, ahol megtalálhattam azokat a könyveket, weboldalakat amelyre szükségem volt a projekt elkészítésében.

Ábrák jegyzéke

Táblázatok jegyzéke

8. fejezet

Könyvészet

8.1. Referenciák

1. https://en.wikipedia.org/wiki/Genetic_algorithm
2. https://en.wikipedia.org/wiki/Pattern_recognition
3. https://en.wikipedia.org/wiki/Hough_transform
4. https://ethereum.org/en/
5. https://en.wikipedia.org/wiki/C%2B%2B
6. https://en.wikipedia.org/wiki/Visual_Studio
7. https://en.wikipedia.org/wiki/OpenCV
8. https://www.opencv-srf.com/2011/09/object-detection-tracking-using-contours.html)
9. https://docs.opencv.org/4.x/d1/dfb/intro.html

8.2. Irodalomjegyzék

1. Ender Ozkan and Chilukure K. Mohan. *Shape recognition using genetic algorithms.* School of Computer and Information Science Syracuse University, Syracuse, June, 1996, NY 13244-4100, U.S.A.
2. Majida Ali Abed, Ahmad Nasser Ismail and Zubadi Maitz Hazi. *Pattern recognition Using Genetic Algorithm.* International Journal of Computer and Electrical Engineering, Vol. 2, No. 3, June, 2010, 1793-8163.
3. Longbin Chen, Julian J. McAuley, Rogerio S. Feris, Tiberio S. Caetano and Matthew Turk *Shape Classification Through Structured Learning of Matching Measures.* IEEE Conference on Computer Vision and Pattern Recognition, August, 2009, 1063-6919.

4. L. Haldurai, T. Madhubala and R. Rajalakshmi . *A Study on Genetic Algorithm and its Applications*, Department of Computer Science (PG), Kongunadu Arts and Science College, Coimbatore, India, October, 2016