# IKKK 5. kutatási főirány

## Beszámoló jelentés

### IRIS, Automatikus raszter-vektor konverziós rendszer fejlesztése

A projekt fejlesztő csapata:


Dezső Balázs,
programtervező
matematikus,
doktorandusz

Máriás Zsigmond,
programtervező
matematikus,
doktorandusz

Elek István,
geofizikus,
egyetemi docens,
projekt vezető



**Budapest, ELTE Informatikai Kar, IKKK**

**2007. november**

# Tartalomjegyzék

# Bevezetés

Az IRIS az Intelligent Rasterimage Interpretation System szavak rövidítése. A projekt célja egy olyan programrendszer létrehozása, amely jó minőségű, automatikus raszter-vektor konverzióra képes.

A térinformatika régi keletű problémája a nagymennyiségű grafikus adatbevitel gyorsítása, esetleges automatizálása. Léteznek különböző szinvonalú, többé kevésbé jól működő automatikusnak mondott megoldások, de valamennyi módszer jellemzője, hogy csak speciálisan kedvező körülmények között képesek valóban hatékonyan dolgozni. Ilyen különlegesen kedvező esetek az olyan papírtérkép nyomatok szkennelt változatai, amelyek csak a szintvonalat tartalmazó rétegről készültek, és ezáltal a raszteres állomány feldolgozása viszonylag könnyen megvalósítható. A hagyományos vektorizáló algoritmusok a vonalkövetés technikáját alkalmazzák, amely a kezdőpontból kiindulva az azonos színű és intenzitású pixelek követésével állítanak elő vonalakat. Ez a logika természetesen nem képes egy olyan bonyolult rendszernek, mint a térkép a feldolgozását elvégezni. A legtöbb esetben csak félautomatikus megoldásokkal találkozhatunk, amelyek addig működnek automatikusan, amíg valamely nehezen átlátható döntési helyzetbe nem kerülnek (vonalkereszteződések, bonyolult csomópontok, stb.), és ott az ember interaktivitása révén képesek csak a megfelelő irányban folytatni a munkát.

A IRIS hároméves futamideje alatt alaposan megismertük a raszter-vektor konverzió problémakörét. A kezdeti elképzeléseinket a fejlesztésről – apróbb részletkérdésektől eltekintve – alapvetően nem kellett megváltoztatnunk, vagyis megvalósíthatónak bizonyult a szkennelt topográfiai térképek vektorizálása. Természetesen nem állítjuk, hogy nincs javítani való a rendszeren (pl. a feliratok elolvasása), de azt állítjuk, hogy a megkezdett úton végig lehet menni, és végül valóban professzionális minőségű vektorizáló szoftver készíthető.

Különösen érdekesnek gondoljuk, hogy a fejlesztés során belefutottunk egy alapkutatási problémába, a tudás reprezentáció kérdéskörébe. A létező módszereket megvizsgáltuk, de nem találtuk alkalmasnak arra, hogy ezekre támaszkodva alkossunk egy valóban intelligens raszter-vektor konverziós térkép értelmező programot. Saját megközelítéssel próbáltunk felvázolni egy lehetséges utat, amely azonban lényegesen hosszabbnak és nehezebbnek látszik, mint a projekt elején gondoltuk, mert akkor még úgy gondoltuk, hogy a meglévő MI módszerek használhatók lesznek.

A projekt a tudásreprezenetáció nélkül is eredményesnek tekinthető, mivel a kitűzött célokat sikerült elérnünk, és a rendszer továbbfejlesztése akár piaci verzió létrehozását is eredményezheti.

# 1. Fejezet: Előfeldolgozás

A topográfiai térképek vektorizálásának kiinduló alapanyaga egy raszteres kép, melyet szkennelés eredményeképp kapunk. A feladat tehát raszter-vektor konverzió, ezekből a képekből kiindulva. Az előfeldolgozás célja ezen belül a képek minél jobb előkészítése a további lépésekhez.

## Szkennelt térképek hibái és forrásaik

A kezdeti kép általában hibákkal terhelt, az előfeldolgozó lépés egyik feladata, hogy ezeket a hibákat kiküszöböljük.

### A nyomtatás és a használat során keletkező hibák

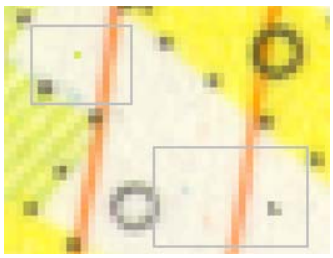
A szkennelt térképeken lévő hibák egy része a térképek nyomtatásából és a használat során keletkezett sérülésekből és hibákból adódik. Ezek a hibák általában „salt and pepper noise" típusúak, azaz nem kívánatos színű pontok a képen. Ezeket digitális szűrőkkel viszonylag egyszerű kezelni, ugyanis a digitális szűrők közül sok pont erre a célra lett kitalálva. Nem minden hiba küszöbölhető ki ilyen egyszerűen azonban.

### A nyomtatás sajátosságaiból és a szkennelésből adódó hibák

A nyomtatás sajátosságairól azt fontos tudni, hogy a térképek úgy készülnek, hogy rétegeket nyomtatnak egymásra. Ebből adódik, hogy az elvileg ugyanolyan színű entitások más-más alapszínre nyomtatva mégsem azonos színűek a papíron. Az emberi szemnek könnyű azonosítani az azonos(nak szánt) színeket, de ugyanezt egy programrendszerben megfogalmazni, algoritmizálni már sokkal nehezebb. Nagyon gyakran előfordul, hogy az ugyanazt jelölő és ugyanolyan(nak szánt) színű jelek színkódja a szkennelt raszteres képen nagyon különbözik.

A szkennelés során a szkennert a legnagyobb színfelbontáson érdemes használni. Ha megpróbálnánk megszorítani mondjuk 16 színre, akkor a szkenner is csak valamilyen - valószínűleg egyszerű, a térképészeti konvenciókat nem ismerő - döntési mechanizmust használna arra, hogy egy intenzitásértéket besoroljon valahova. Nekünk viszont a célunk az, hogy magunk kontrolláljuk ezt a lépést. A pixelek besorolása egy osztályba az egész térképvektorizálás során visszatérő kérdés: nem csak a kiinduló színintenzitásra alapozunk, ami egy réteghez tartozik, hanem később azt is szeretnénk, ha egy pozíciónak a térképi rétegek szerint egyenként megmondanánk a funkcióját.

Tehát a szkennelés eredményeképp olyan raszteres képet kapunk, amin a térkép készítésekor azonos színűnek szánt képpontok színkódja messze eshet egymástól.



*1.1: Példa a nyomtatásból és szkennelésből adódó színhibákra. Az elméletileg ugyanolyan színű entitások egymás mellett láthatóan nagyon különböznek 1.2: Só és bors típusú hibák.*

## A nyersvektorizáláshoz szükséges képek előállítása

### A vektorizálás feladatai

A raszteres állományon a vektorizálás során két feladatunk van:

- egyrészt a nagy foltok, területek felismerése, ezek topológiailag helyes síkbeli interpretációja
- másrészt a térképi jelkulcsok, szintvonalak, utak, épületek és egyéb entitások felismerése és geometriai interpretációja

### Az előfeldolgozás céljai

Az előfeldolgozási lépés egyik célja, hogy képszűrési eljárásokkal elősegítsük a nagy területek szegmentálását és vektorizálását, valamint az apró, finom részletek minőségét javítsuk, hogy a mintaillesztés során minél egyértelműbben azonosíthatóak legyenek.

A topográfiai térképek esetén felhasználhatjuk azt a tudást, hogy ezek megfelelnek bizonyos térképészeti konvencióknak. Ezek közül az egyik legfontosabb, hogy tudjuk milyen színek fordulhatnak elő, illetve, hogy azok mit jelenthetnek. A nagy területek sosem lehetnek például piros színűek, a szintvonalak egyszínűek és pár pixel vastagságúak, stb. A színek térképeknél játszott fontok szerepe, mögöttes tartalma miatt szükség van olyan eljárásokra, melyek bizonyos színeket azonosítanak.

Az előfeldolgozási eljárás céljai ellentmondanak egymásnak, hiszen a nagy területek felismerése például jóval egyszerűbb, ha a jelkulcsokat valamilyen szűréssel teljesen leszedtük. Így tehát az előfeldolgozási lépés során a szűrési eljárások több csoportjára van szükség.

### Részletek eltávolítása

Amikor a nagy foltokat, területeket azonosítjuk, olyankor a jelkulcsokat,

szintvonalakat, utakat, rácsokat, feliratokat meg kell próbálnunk a lehető legjobban eltüntetni, a hozzájuk tartozó pixeleket arra a színre cserélni, amin ők fekszenek. Ezekhez a lépésekhez különböző simító- és statisztikai szűrők él- és sarokmegőrző változatai, szegmentálási eljárások szükségesek

**Részletek élesítése, minőségük javítása**

Amikor a szintvonalakat, a jelkulcsokat azonosítjuk, olyankor ezeket a részleteket élesíteni, kontrasztosítani kell. Ehhez hisztogram alapú szűrésekre, színek azonosítására, szegmentálási eljárásokra van szükség.

Ebben a fejezetben áttekintjük a digitális képelemzés azon algoritmusait, amelyek az előfeldolgozás során szóba jöhetnek, valamint azokat a térképészeti konvenciókra alapuló heurisztikákat, melyek a fenti célokat jól szolgáló szűrőket eredményeznek, kitérve a hatékonysági kérdésekre és a szűrők eredményének elemzésére is.

A fejezet végén az eljárásokból szervezett munkafolyamatok és azok eredményeinek elemzése következik.

# Képek kezelése

A képfeldolgozás során többféle képtípusra is szükségünk van, amelyeket lehetőleg ugyanúgy szeretnénk kezelni.

A képfeldolgozási algoritmusok bemenete illetve segédváltozója lehet:

- színes kép, amely pixeljei RGB színkódolásban adottak, az egyes komponenseket szürkeárnyalatos képeknek tekintve színcsatornának vagy sávnak nevezzük
- szürkeárnyalatos kép, mely minden képponthoz egyetlen intenzitásértéket tárol a [0..255] intervallumból
- bitkép, mely minden egyes képponthoz egy logikai értéket tárol
- racionális értékeket tartalmazó kép, mely minden képponthoz egy racionális értéket tartalmaz

A szűrési algoritmusok egy része értelmezhető minden képtípusra, más része csak bizonyos képeken értelmezhető. Előbbire példa a medián szűrő, utóbbira az élvékonyító szűrő, mely csak bitképeken működik.

### Képek tárolása

Mivel a különböző képtípusok megvalósítására a legkézenfekvőbb mód osztálysablonok készítése C++ nyelven. Az osztálysablonok segítségével az osztály kellőképp általános lesz, a C++ nyelvre pedig hatékonysági okokból esett a választás.

A képosztály két paramétert kap a példányosítás során, az egyik azt mondja meg, hogy hány sávos a kép, a második pedig azt, hogy egy sáv valamelyik képpontját hány biten ábrázoljuk. Így egy <3,8> paraméterekkel példányosított osztálysablon egy színes képet ír le, <1,1> paraméterek esetén pedig egy bitképet. A képosztályokhoz megfelelő pixelosztály is készült, amely ugyanezeket a sablonparamétereket, ugyanezzel a jelentéssel kapja. A képeket egy vektorban, sorfolytonosan tároljuk.

## Egyszerű áttérés képtípusok közt

Gyakran szükség van a különböző képtípusok közti konverzióra, például azoknál a szűrési eljárásoknál, amelyek nem értelmezettek minden képtípusra.

### Áttérés színes képről szürkeárnyalatosra

A színes képről szürkeárnyalatosra való áttérés során a kép sávjainak valamilyen súlyozott átlagát számoljuk. A leggyakoribb súlyozás a 0.3R, 0.59G, 0.11B[1]. Ezeket a súlyokat az magyarázza, hogy az emberi szem a zöld színre a legérzékenyebb, majd a pirosra és végül a kékre. A szürkeárnyalatossá alakításnak más módszerei is elképzelhetőek, például a

1 William K. Pratt, Digital Image Processing, John Wiley & Sons, New York, 1991.

$$\frac{max(R,G,B)+min(R,G,B)}{2}$$

képlettel számolva a HLS színmodell lightness komponensével is számolhatjuk a szín szürkeárnyalatossá alakított változatát.

**Áttérés szürkeárnyalatos képről bitképre, egyszerű küszöbölés.**

A szürkeárnyalatosról bitképre térés tulajdonképpen a pixelek két osztályba való sorolása, vagyis egy egyszerű szegmentálási eljárás. Ez sokféleképp lehetséges, a legegyszerűbb módszer az, hogy egy paraméterben adott értéknél világosabb pixeleket igazra, a sötétebbeket hamisra állítjuk az eredményben.

Az összetett küszöbölési algoritmusok célja, ennek küszöbértéknek a kiszámolása, lehetőleg úgy, hogy az egy jó osztályozást adjon. A jó osztályozáson azt értjük, hogy a képen a különböző természetű dolgok lehetőleg különüljenek el. A dolgozatban az összetett szűrési eljárásoknál tárgyalom Otsu algoritmusát a küszöbérték megtalálására.

# Digitális szűrési eljárások

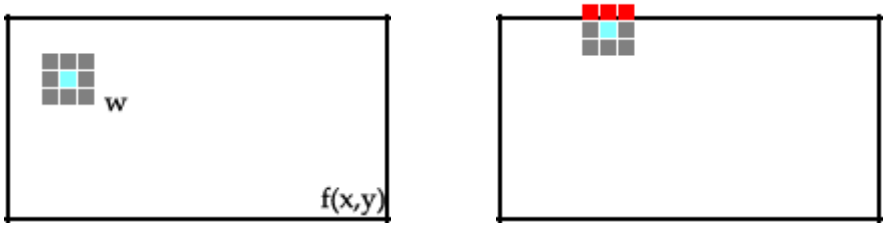
A digitális szűrők azok a képfeldolgozó algoritmusok, amelyek alapvetően úgy működnek, hogy egy kernelt - *2k+1 x 2k+1* méretű ablakot – futtatnak végig egy kép minden pontján, úgy, hogy az aktuális képpont a kernel közepére esik, majd a kernel alá eső értékekből valamilyen eljárással kiszámolják az aktuális pixel új értékét:

$$g(x,y)=F[f,(x,y)]$$

Ebben a képletben a *g(x,y)* jelenti az új kép *(x,y)* koordinátájú pontját, az *f* az eredeti képet, az *F[]* pedig azt az operátort, ami az kép *(x,y)* koordinátájának szomszédságából kiszámolja az új intenzitásértéket.

Ezek a szűrők azt a megfigyelést használják ki, hogy az egymáshoz közeli pixelek értékei jobban összefüggenek, mint az egymástól távoli pixelek. Ezeknek a szűrőknek másik fontos jellemzője, hogy nem rekurzívak. Ez azt jelenti, hogy az eljárás csak az eredeti intenzitásértékektől függ, azaz mindig az eredeti képből vesszük a képpont szomszédságát.

Mivel a széleknél a kernelmátrix „lelóg” a képről, ezeket a pixeleket ugyanolyan intenzitásúnak vesszük, mint a hozzá legközelebb - a kép szélén - lévő pixeleket.




*1.3 Általános, 3x3-as méretű szűrő. 1.4: A kép szélein a lelógó pixeleket a legközelebbi, képbe beleeső pixellel helyettesítjük.*

## Konvolúciós szűrők

A konvolúciós, vagy lineáris szűrők úgy működnek, hogy egy kernelmátrixban szereplő értékekkel súlyozott átlagaként számolják ki az új intenzitásértéket.

$$\sum_{i=x-k\,j=y-k}^{i=x+k\,j=y+k} f(i,j)*w(i+k,j+k) \qquad (1)$$

## Szeparábilis szűrők

Ha kernelmátrix speciális alakú, akkor a konvolúció végeredményét jóval gyorsabban is számolhatjuk. Ha fennáll, hogy a *w* kernelmátrix felbontható egy

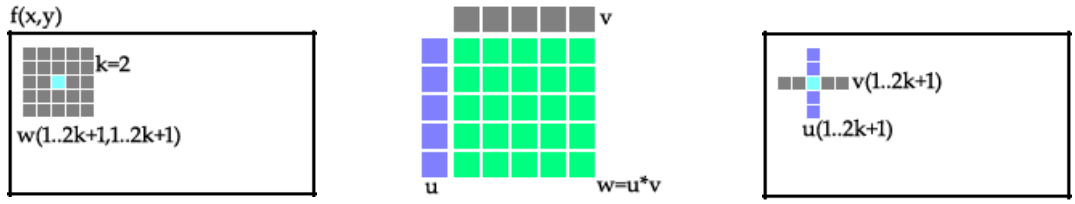
oszlop- és egy sorvektor szorzatára, azaz:

$$w(i,j)=u(i)*v(j)$$

akkor megtehetjük azt, hogy a először kiszámoljuk a kép *u*-val vett konvolúcióját, majd az eredmény *v*-vel vett konvolúcióját, azaz:

$$f'(x,y)=\sum_{i=x-k}^{i=x+k} f(i,y)*u(i+k) \quad \text{és ebből}$$

$$g(x,y)=\sum_{j=y-k}^{j=y+k} f'(x,j)*v(j+k)$$




*1.5: általános konvolúciós szűrés, k=2 esetén 5x5 méretű kernellel. 1.6 w[5x5] méretű szeparábilis szűrő, előáll u és v szorzataként. 1.7: a vektormaszkok alkalmazása a képen, természetesen nem egyszerre, hanem egymás után.*

Így a konvolúció kiszámításának műveletigénye ***O****(n*m*(2k+1))*. Mivel a kernel mérete általában kicsi, ezért ez értelmezhető lineárisnak a kép pixeljeinek számában. A művelethez ***O****(n*m)* segédmemória szükséges.

**Box szűrő**

A box szűrő olyan speciális kernelű szűrő, ahol a kernelmátrixban szereplő összes érték ugyanannyi, vagyis a kernel alá eső pixeleket átlagoljuk. A box szűrő hatása a kép simítása. Önmagában nem ad túl jó eredményt, de élmegőrző változata több helyen is használható a munkafolyamatban

**Gauss szűrő**

A Gauss szűrő kernelében az értékek a kétdimenziós Gauss eloszlás értékei szerepelnek. A Gauss eloszlás a következőképp számolható:

$$G(x,y)=\frac{1}{2\pi\sigma^2}e^{-\frac{(x^2+y^2)}{2\sigma^2}}$$

Így a *0* várható értékű, *σ* szórású Gauss haranggörbét kapunk, amelyből kernelt úgy

számolunk, hogy a rácspontokon mintavételezzük a G függvényt. Mivel igaz, hogy

$$\frac{1}{2\pi\sigma^2} e^{-\frac{(x^2+y^2)}{2\sigma^2}} = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}} * \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{y^2}{2\sigma^2}}$$

ezért a Gauss szűrő is szeparábilis, az u és v vektor a következőképp számolható:

$$u(x) = v(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-(k+1))^2}{2\sigma^2}}$$

A Gauss szűrőnek létezik egy speciális, még gyorsabb implementációja. Ha a haranggörbe értékeit 2 hatványokkal közelítjük, akkor nem kell szoroznunk amikor a konvolúciót végezzük, hanem megfelelő számú bittel kell csak eltolni az értékeket. Ekkor az u és v vektorok a következőképp alakulnak:

$$u(x) = v(x) = 2^{k+1-|x-k|}$$

A Gauss szűrő hatása hasonló a box szűrőhöz, azaz simító szűrő. A simítás mértéke és a szórás nagyságától függ, természetesen nagy szórás esetén a kernel méretét is növelni kell, hogy tényleg azt a hatást érjük el, amit szeretnénk. A Gauss szűrőnek szintén nem önmagában, hanem más algoritmusokban van szerepe, pl. a Canny-féle éldetektorban használjuk.

A Gauss szűrő segítségével van lehetséges a képek méretének egyszerű csökkentésére (felezésére). Az algoritmus úgy működik, hogy a kiinduló képre alkalmazzuk a Gauss szűrőt, majd elhagyjuk minden második sort és oszlopot. Az így létrejövő folyamatosan feleződő képeket Gauss-piramisnak is nevezik. A Gauss-piramisban az egymás feletti képeket egymásból kivonva a heterogén részek felerősödnek, ez a tulajdonság jól használható a textúrák detektálásánál.

*1.8: Eredeti kép. 1.9:A kép 3x3-as, Gauss-szűrt változata. σ=1. 1.10: A kép 7x7-es, Box szűrt változata. 1.11: A kép 7x7-es Gauss szűrt változata. σ =2.*

## Nemszeparábilis szűrők

A nem szeparábilis szűrők azok, melyek kernelmátrixa nem írható fel két vektor szorzataként. Ilyenkor az eredeti (1) képletben szereplő összegzést kell

megvalósítani. Ennek műveletigénye $O(n*m*(2k+1)^2)$, és $O(n*m)$ segédmemória szükséges.

Ha 3x3-as méretű kernellel dolgozunk, akkor a segédmemóriát le lehet szorítani $O(n)$-re, úgy, hogy 3 tömbben tároljuk az előző, az aktuális és a következő sort. Így a kiszámolt értékeket rögtön az eredeti képre írjuk, a következő sornál pedig eggyel eltoljuk a tömböket és beolvassak a következő sort. Ez általánosítható $2k+1$ méretű szűrőkre is

**Gradiens szűrő**

A gradiens szűrő használatával a kép, mint felület, pontjaiban vett deriváltak x és y irányú gradiensét közelítjük a különbségi hányadossal. A nagy intenzitásváltozásokra reagál, a homogén területekre 0-t ad eredményül.

A gradiens szűrő kernelje a következőképp alakul:

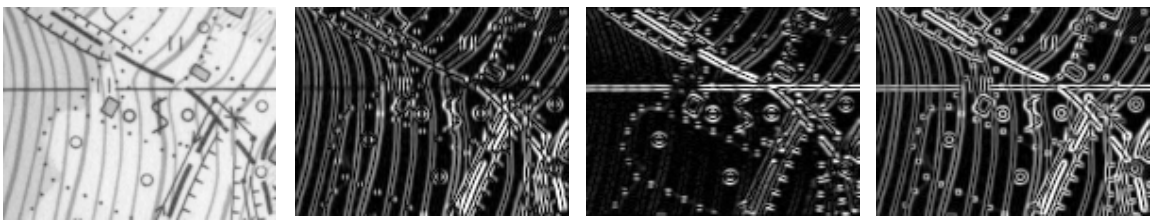
$G_x =$

| | | |
|---|---|---|
| -1 | 0 | 1 |
| 2-p | 0 | p-2 |
| -1 | 0 | 1 |

$G_y =$

| | | |
|---|---|---|
| -1 | 2-p | -1 |
| 0 | 0 | 0 |
| 1 | p-2 | 1 |

A p érték szabadon választható, gyakran használt értékek a p=2, p=3, p=2+√2, első esetben Prewitt-, második esetben Sobel, harmadik esetben izotropikus operátorról beszélünk. Az eredményt p-vel normalizálni kell.



*1.12: Eredeti kép. 1.13-14: A kép x és y irányú gradienseinek közelítése izotropikus gradiens szűrővel. 1.15: Az előző két képből számolt élkiterjedés.*

**Laplace szűrő**

A Laplace operátor definíciója:

$$L(f(x,y)) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

Ha a Laplace operátort diszkretizáljuk, akkor a következő egyenletet kapjuk a pontbeli második deriváltra:

$$f''(x,y) = f(x-1,y) + f(x+1,y) + f(x,y+1) + f(x,y-1) - 4*f(x,y)$$

Így a Laplace operátor értékét az (x,y) pontban a következő konvolúciós kernellel számolhatjuk:

| | | |
|---|---|---|
| 0 | 1 | 0 |
| 1 | -4 | 1 |
| 0 | 1 | 0 |

A Laplace szűrő gyakorlatilag a kép simított és eredeti változatának különbsége, ezért az intenzitásváltozásokra, így a hibákra is nagyon erősen reagál, önmagában nem túl hatásos.

**LoG szűrő**

A Laplace szűrőt Gauss szűrő után alkalmazva éldetektálásra alkalmazható. Mivel a konvolúció asszociatív, ezért a megtehetjük, hogy a Laplace és Gauss szűrő kernelét konvolváljuk és az eredményként kapott mátrixszal dolgozunk. Ezt a szűrőt szokták „Laplacian of Gaussian" - LoG - is nevezni. A LoG szűrő kevésbé érzékeny a zajra, jó éldetektor. A kernelmátrix a következőképp számolható:

$$\mathrm{LoG}(x,y) = \frac{-1}{\pi\sigma^4} * \left[1 - \frac{x^2+y^2}{2\pi\sigma^2}\right] e^{-\frac{(x^2+y^2)}{2\sigma^2}}$$

**Emboss szűrő**

Az emboss szűrők célja az intenzitásváltozások detektálása. Ehhez olyan kernelt alkalmaz, amelynek két átellenes szélében +1 illetve -1 található. Attól függően, hogy milyen irányú átlóban vannak az értékek, az arra merőleges élekre reagál érzékenyen a szűrő. Példa egy emboss szűrőre:

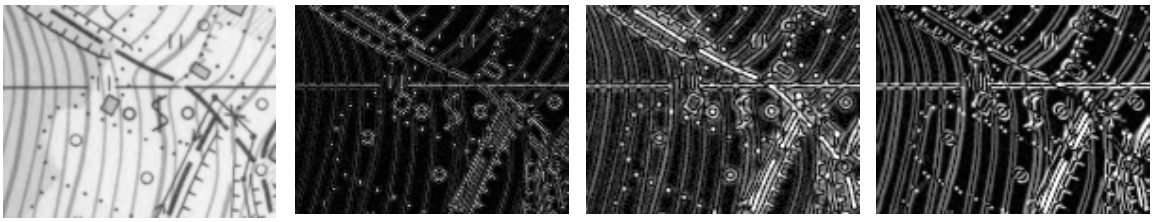
| | | |
|---|---|---|
| 1 | 0 | 0 |
| 0 | 0 | 0 |
| 0 | 0 | -1 |

**Kép élesítése**

Az eddig tárgyalt szűrők segítségével lehetséges a képek élesítése, minőségének javítása. Működésének alapelve, hogy a kép eredeti simított változatának különbségét hozzáadja az eredeti képhez:

$$g(x,y) = f(x,y) + (f(x,y) - f_{mean}(x,y))$$

ahol az $f_{mean}$ jelenti a simító szűrő eredményét. Ez lehet Gauss, box, stb. típusú simító szűrő.



*1.16: Az eredeti kép. 1.17: A kép Laplace szűrt változata. 1.18: A kép LoG szűrt változata. 1.19: A kép emboss szűrt változata: ÉK-DNY irányú élek megtalálására beállítva.*

## Nemlineáris szűrők

A nemlineáris digitális szűrők közé azokat az eljárásokat soroljuk, amelyek ugyanúgy egy pixel szomszédsága alapján számolják ki az új értékét, de ezt nem valamilyen lineáris kombinációként teszik, hanem egyéb módszerekkel. Itt nem beszélünk kernelmátrixról, hanem csak kernelről, vagy kernelablakról.

### Rank szűrők, medián szűrő

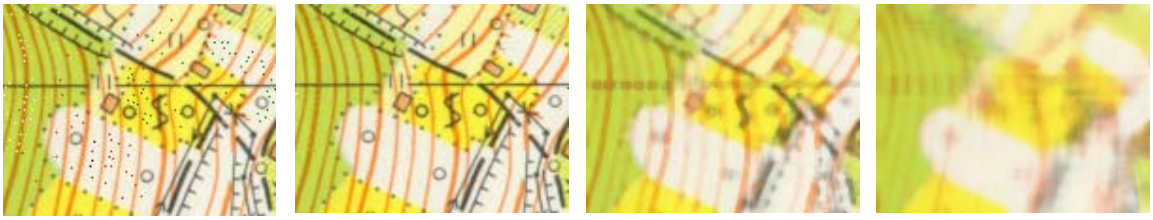
A rank szűrők alapvető ötlete az, hogy a kernelablak alá eső pixelek intenzitásértékeit állítsuk nagyság szerint sorba, majd ez alapján a sorrend alapján válasszunk új intenzitásértéket a pixelünknek. A leggyakrabban használt rank szűrő a medián szűrő, mely a nagyság szerint középső értéket választja ki. Ez egy simító szűrő lesz, mely a lokális zajokat nagyon jól eliminálja. A „salt and pepper" típusú hibákat nagyon jól eltünteti, mert amikor egy ilyen pixelhez érünk, a környező pontok színe a sorrend szélére tolja ki a nemkívánatos hibákat.

A medián szűrő fontos tulajdonsága, hogy 2k+1 méretű kernelméret esetén a k-nál vékonyabb vonalakat eltünteti a képről. Ez kívánatos eredmény, amikor a nagy területeket próbáljuk kiemelni. Sajnos az éleket eltolhatja és a sarkokat lekerekíti, de az algoritmus kiegészíthető úgy, hogy megőrizze őket.

A rank szűrők direkt implementációjának műveletigénye $\boldsymbol{O}(n*m*(2k+1)^2*\log(2k+1))$, ami abból adódik, hogy minden pont körül rendezni kell $(2k+1)^2$ darab pontot. Ha a k-adik elem kiválasztás algoritmusát használjuk, akkor a műveletigény $\boldsymbol{O}(n*m*(2k+1))$ lesz.

### Konzervatív simítás

A konzervatív simítás zajszűrő eljárás, mely leginkább a „salt and pepper" típusú zajt képes leszedni. Stratégiája, hogy a kernelablakba eső pixeleket nagyság szerint sorba rendezi az aktuális pixel kivételével. Így kapunk egy *[min..max]* intervallumot, és megnézzük hogy az aktuális pixel ebbe az intervallumba esik-e. Ha igen, akkor nem változtatunk a pixel intenzitásán, hogyha a maximumnál nagyobb, akkor az új érték a maximum lesz, ha a minimum alá, akkor pedig a minimum.



*1.17: Az eredeti kép, „só és bors" típusú hibákkal terhelve. 1.18: Konzervatív simítással eltüntetve a hibák. 1.18: 5x5 méretű mediánszűrővel tisztítva a képet. 1.19: 11x11 méretű mediánszűrővel szűrt változtat.*

**Kuwahara szűrő**

A Kuwahara-féle[2] szűrési eljárás zajszűrő, simító hatású. Fontos tulajdonsága, hogy az éleket megtartja, nem tolja, vagy mossa el. Az alapötlet az, hogy a kernelablakot négy átfedő *(k+1)x(k+1)* méretű részre osztja a következő módon:

| | | |
|---|---|---|
| a | ab | b |
| ac | abcd | bd |
| c | cd | d |

A betűjelek azonosítják azt, hogy a pixelek hova tartoznak. A középső pixel mindben benne van. Miután a pixeleket besoroljuk a négy halmazba, kiszámoljuk mindegyik halmazban a pixelértékek átlagát és a tapasztalati szórást. Ezután annak a halmaznak a átlagát adjuk értékül a pixelnek, amelyikben a legkisebb a szórás.

Ez a szűrő élmegőrző lesz, hiszen ha az ablak középpontja egy élen egyik oldalán, az biztosan nem a túloldalból fog értéket kapni, mert a szórást megnöveli az, hogy az aktuális pixel intenzitása nagyon különbözik a túloldalon lévő pixelek intenzitásától, így nagyon megnöveli a szórást.

*1.20: Az eredeti kép. 1.21: A kép 5x5 méretű Kuwahara szűrővel szűrt változata. 1.22: A kép 11x11 méretű Kuwahara szűrővel szűrt változata. 1.23: Az 1.21 képet újraszűrve 5x5 méretű Kuwahara szűrővel.*

A szűrő műveletigénye ***O****(n*m*(k+1))* az átlag és szórásszámítás miatt nagy konstanssal.

2 M. Kuwahara, K. Hachimura, S. Eiho, and M. Kinoshita. Digital Processing of Biomedical Images.,Plenum Press, New York, NY, 1976

## Hisztogram alapú szűrés

A hisztogram alapú szűrések közös tulajdonsága, hogy nem a közvetlenül kép pixeljei, hanem a hisztogram alapján dolgozik a transzformáció. A kép hisztogramja az egyes intenzitásértékek relatív gyakoriságait tartalmazza.

### Hisztogramkiegyenlítés

A hisztogram kiegyenlítésének célja a kép kontrasztosabbá tétele. Ezt úgy éri el, hogy megnézi a hisztogram legnagyobb és legkisebb értékét, majd a pixeleket a minimummal a nulla felé tolja és felszorozza, hogy az intenzitásskála a *[0..255]* intervallumot kitöltse:

$$g(x,y)=\frac{(f(x,y)-min)}{max-min}*255$$

*1.24: Eredeti kép. 1.25: A kép hisztogram normalizálás után. 1.26: Kevéssé kontrasztos, szürkeárnyalatos kép. 1.27: A kép hisztogram normalizálás után.*

### Küszöbölés

A küszöbölés a szürkeárnyalatos képek szegmentálásának egyik módja. Ilyenkor megadunk néhány küszöbértéket, amelyek intervallumhatárokat fognak jelölni. A küszöbölés speciális esete a kétszintes küszöbölés, ami azt jelenti, hogy egyetlen küszöbértéket adunk meg, így a pixeleket két osztályba soroljuk. A küszöbérték meghatározására több stratégia létezik, attól függően, hogy milyen cél lebeg a szemünk előtt, mi a mértéke annak, hogy mennyire jó egy küszöbérték. Általában azt szeretnénk, ha a képen az objektumok jól eltérjenek a hátterüktől. Mivel két osztályba sorolhatunk be minden pixelt, ezért ez nem sikerülhet mindig tökéletesen, de az a jó küszöbölés ezt a lehető legjobban közelíti.

### Otsu-féle küszöbölés[3]

Otsu a jó küszöbölést úgy határozza meg, hogy az a jó osztályozási eredmény, ha a két osztály közti szórás a lehető legnagyobb. Ehhez kiszámolja a kép pixeljeinek tapasztalati várható értékét és szórásnégyzetét.

3 N. Otsu, A threshold selection method from gray-level histograms, IEEE Trans. Sys., Man., Cyber., vol. 9, 1979

$$\mu=\sum_{i=0}^{255} i*P(i) \quad , \quad \sigma^2=\sum_{i=0}^{255} (i-\mu)^2*P(i)$$

Egy t küszöbérték mellett az egyes osztályokon belüli szórása és várható értéke

$$\mu_1(t)=\frac{1}{q_1(t)}*\sum_{i=0}^{t} i*P(i) \quad , \quad \sigma_1^2(t)=\sum_{i=0}^{t} (i-\mu_1)^2*P(i) \quad \text{, illetve}$$

$$\mu_2(t)=\frac{1}{q_2(t)}*\sum_{i=t+1}^{255} i*P(i) \quad , \quad \sigma_2^2(t)=\sum_{i=t+1}^{255} (i-\mu_2)^2*P(i) \quad \text{, ahol}$$

$$q_1(t)=\sum_{i=0}^{t} P(i) \quad , \quad q_2(t)=\sum_{i=t+1}^{255} P(i) \quad .$$

Az osztályokon belüli szórás a két osztály szórásának súlyozott összege, azaz

$$\sigma_w^2(t)=q_1(t)*\sigma_1^2(t)+q_2(t)*\sigma_2^2(t)$$

Az osztályok közti szórást a következőképp definiáljuk

$$\sigma^2=\sigma_w^2(t)+\sigma_b^2(t) \quad \text{, vagyis} \quad \sigma_b^2(t)=\sigma^2-\sigma_w^2(t) \quad \text{, amit kifejtve az kapjuk, hogy}$$

$$\sigma_b^2(t)=q_1(t)*q_2(t)(\mu_1(t)-\mu_2(t))^2=q_1(t)*(1-q_1(t))(\mu_1(t)-\mu_2(t))^2$$

Az optimális szórás számunkra az, ami a lehető legjobban elkülöníti az osztályokat. Mivel a kétféle szórás összege egyenlő a teljes szórással, ezért két, egymással ekvivalens célunk lehet az optimum megtalálásához: minimalizáljuk az osztályokon belüli szórást, vagy maximalizáljuk az osztályok köztit. Ha az utóbbit választjuk, akkor az egyes t értékekre a $q_1(t+1)$, $\mu_1(t+1)$, $\mu_2(t+1)$ értékeket számolhatjuk a $q_1(t)$, $\mu_1(t)$, $\mu_2(t)$ értékek felhasználásával:

$$, \quad q_1(0)=0$$

$$\mu_1(t+1)=\frac{q_1(t)\mu_1(t)+(t+1)P(t+1)}{q_1(t+1)} \quad , \quad \mu_1(0)=0$$

$$\mu_2(t+1)=\frac{\mu-q_1(t+1)(\mu_1+1)}{1-q_1(t+1)}$$

Így a következő algoritmust kapjuk:

1) számoljuk ki P-t, μ-t és σ-t
2) t=0-tól 255-ig számoljuk ki minden értékre számoljuk ki $q_1(t)$, $\mu_1(t)$, $\mu_2(t)$ értékeket, majd ebből a $\sigma_b^2$ értéket
3) válasszuk $t_{opt}$-nak az $argmax(\sigma_b^2)$-t

Az Otsu-féle küszöbölés általánosítható, azaz több szintű küszöbölést is lehet ezzel a stratégiával előállítani[4]. A vázolt eljárás műveletigénye ***O****(n*m)*.

*1.28: Eredeti szürkeárnyalatos kép. 1.29: A kép Otsu eljárásával küszöbölt változata. 1.30-31: A kép halványabb változatát az eljárás ugyanolyan jól küszöböli. Egy előre beállított küszöbérték az első képet jól, a másodikat teljesen rosszul küszöbölte volna.*

Az Otsu-féle küszöbölés nagyon hasznos az előfeldolgozás során, elsősorban a színek detektálásánál használható, mint a döntési mechanizmust támogató egyik segédalgoritmus.

## Összetett szűrések

Az összetett szűrések közé azok azok az általános digitális képelemzésbeli eljárások kerültek, amelyek több szűrő együttes eredményeképp állítanak elő olyan képet, amely az eredeti kép valamilyen tulajdonságát emeli ki. Ilyen eljárásból sokféle létezik, de a térképvektorizáshoz a pontos éldetektálás a vonalvékonyítás algoritmusa hasznos, ezért a következőkben ezeket tárgyalom.

### Canny éldetektor

A Canny-féle éldetektor[5] olyan élfelismerést valósít meg, amely a következő jó tulajdonságokat egyesíti: nem érzékeny az élek állására, minden élet helyesen detektál és egy élet egy vonallal rajzol meg. Ehhez a következő értékeket használja fel:

$$grad\, f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right) = (f_x, f_y)$$ a kép gradiense az (x,y) pontban

$$M(x, y) = \sqrt{f_x^2 + f_y^2}$$ az „él erőssége” az (x,y) pontban

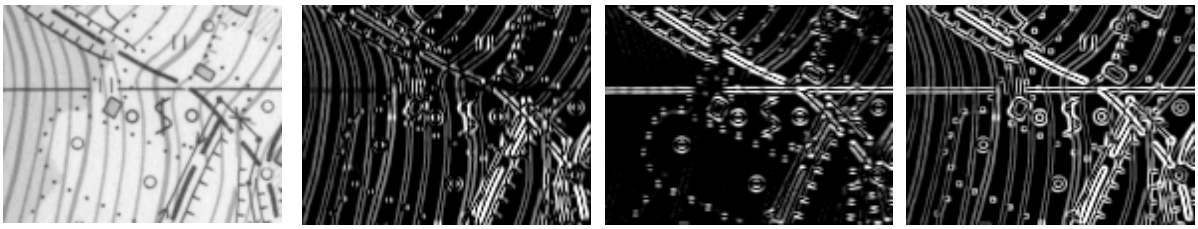
4 P. Liao, T. Chen and P. Chung, A Fast Algorithm for Multilevel Thresholding, Journal of Information Science and engineering 17 2001.

5 Canny, J., A Computational Approach To Edge Detection, IEEE Trans. Pattern Analysis and Machine Intelligence, 8:679-714, 1986

$$\Theta(x, y) = \arctan\left(\frac{f_x}{f_y}\right)$$ egy pontban vett derivált irányvektora, az él normálvektora.

Az $f_x$, $f_y$ értékeket közelíthetjük úgy, hogy az x és y irányú izotropikus gradiens szűrővel vett konvolúcióját számoljuk a képnek az adott pontban, majd ebből kiszámoljuk minden pontban az M(x,y) értékeket. Ezen a képen az élek látszanak, de minden él több pixel vastag, hiszen a képeken az élek általában nem tökéletesek, valamint ezzel a módszerrel még egy tökéletes él szomszédságában is 0-nál nagyobb értékeket kapunk.

Azt a problémát oldja meg a nem-maximális élek kiküszöbölése[6]. Ezt úgy tehetjük meg, hogy az M(x,y)-t lemásoljuk egy kimeneti képbe, a Θ(x,y)-ban adott normálvektor által meghatározott szögben lépünk mindkét irányba az M(x,y) képen, és ha bármelyik intenzitásérték nagyobb az aktuálisnál, akkor töröljük a pixelt a kimeneti képen. A nem-maximális élek kiküszöbölésének eredményeképp egy olyan képet kapunk, amelyen minden él rajta van és mindegyik egyetlen vonalként jelenik meg.



*1.32: Az eredeti kép. 1.33-34: A kép x és y irányú gradiensének közelítése a gradiens szűrő segítségével. 1.35: Az előző két képből előállított élkiterjedés.*

Mivel így minden él, még a leggyengébbek is rajta lesznek a kimeneti képen, szükségünk lehet arra, hogy a gyengébb éleket eltüntessük és az erősebb, globális éleket tartsuk meg. Ha valamilyen küszöbölési eljárást használnánk, akkor az nem lenne tekintettel az élek folytonosságára, mi pedig nem szeretnénk, ha az élek megszakadnának. Erre a megoldást az élküszöbölés jelenti. Ennek alapötlete, hogy egy határ alatt minden pontot hagyjunk el, egy határ fölött mindent tartsunk meg, a köztes pixeleket pedig aszerint tartsunk meg, hogy eljutunk-e belőle biztosan jó pixelbe köztes pontokon. Ehhez a legjobb, ha mélységi bejárás egy változatát implementáljuk, kiegészítve azzal, hogy egy biztosan jó pontba érve az egész útvonalon megtartjuk a pixeleket.

6 Lindeberg, Tony "Edge detection and ridge detection with automatic scale selection", International Journal of Computer Vision, 30, 2, pp 117--154, 1998.

*1.36: Az élkiterjedést tartalmazó kép. 1.37: A nem-maximális éleket kiküszöbölve kapott kép. 1.38-39: Az élek küszöbölése, 10, 40 illetve 40, 100 paraméterekkel. A második esetben csak a legerősebb élek maradtak meg.*

**Élvékonyítás**

A vékonyítási eljárások célja, hogy a képen lévő objektumoknak a vonalas vázát állítsa elő. Fontos tulajdonsága, hogy az objektumok topológiája megmarad és az objektum végpontjait nem törli. Az élvékonyító eljárások általában úgy működnek, hogy az objektum széléről addig hámoznak le pixeleket, amíg végül egy pixel szélességű vonalakká nem alakítja. Az élvékonyítás speciálisan bitképeken dolgozik.

A következőkben bemutatott algoritmus[7] a 4-es összefüggőség megtartásával oldja meg a feladatot. Vezessük be a p1 aktuális pont szomszédságára az alábbi jelölést:

| | | |
|---|---|---|
| $p_3$ | $p_2$ | $p_9$ |
| $p_4$ | $p_1$ | $p_8$ |
| $p_5$ | $p_6$ | $p_7$ |

Ezen kívül vezessük be a TR($p_1$) függvényt, amely a p1 körül a 0-ból 1-be való átmeneteket számolja a pixeleket az alábbi sorrendben véve: {$p_2$, $p_3$, $p_4$, $p_5$, $p_6$, $p_7$, $p_8$, $p_9$}, valamint a NZ($p_1$) függvényt, ami azt mondja meg, hogy hány nem nulla pixel van a $p_1$ körül.

Az algoritmus minden iterációjában töröljünk egy aktuális $p_1$ pixelt, ha igazak rá az alábbi feltételek:

- $2 \leqslant NZ(p_1) \leqslant 6$
- $TR(p_1) = 1$
- $p_2 * p_4 * p_8 = 0 \vee TR(p_2) \neq 0$
- $p_2 * p_4 * p_6 = 0 \vee TR(p_4) \neq 0$

Álljunk meg abban az iterációban, amikor nem történik változás.

7 Lam L., Lee S. W., Suen C. Y. (1992): Thinning methodologies - a comprehensive survey. IEEE Trans. on Pattern Analysis and Machine Intelligence, Vol. 14, No. 9, pp. 869-887.

A TR(p) függvény hasznos a vékonyított kép vizsgálata során is, a „jelváltások" számának vizsgálatával azonosítani lehet a vonalak végpontjait. Az algoritmus igazából a $p_1$ pont 4x4-es környezetével dolgozik, ugyanis ki kell számolni $p_2$ és $p_4$ pont környezetében is a TR függvény értékét.

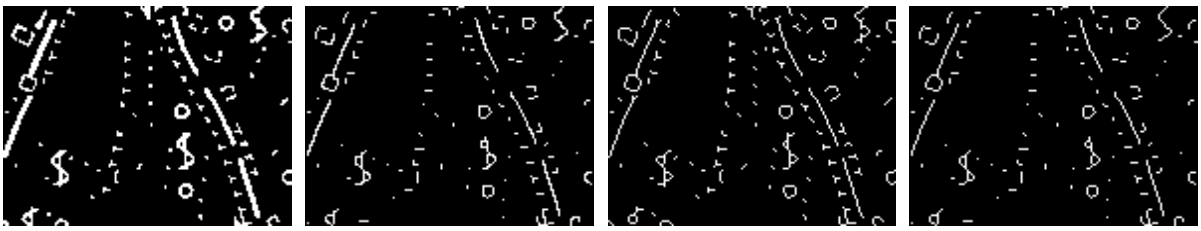
**Morfológiai élvékonyítás**

A morfológiai élvékonyítás[8] ugyanúgy bitképeken dolgozik, de egy másik stratégiát követ. Minden lépésben egy maszkot és lehetséges elforgatásait illeszti a kép pontjaira. A maszknak háromféle pixelértéke van, háttér, objektum és figyelmen kívül hagyott. Ezt akkor fogadjuk el a kép egy pozícióján, ha a maszk objektum pontjai alatt igaz, háttér pontjai alatt hamis, figyelmen kívül hagyott pontjai alatt pedig tetszőleges érték szerepelhet a képen. A maszk a következőképp alakul:

| | | |
|---|---|---|
| 0 | 0 | 0 |
| ? | 1 | ? |
| 1 | 1 | 1 |

| | | |
|---|---|---|
| ? | 1 | 0 |
| 0 | 1 | 1 |
| 0 | 0 | ? |

Ezen a két maszkon kívül még ezek 90°, 180° és 270°-al vett elforgatottjait, tehát összesen 8 maszkot használunk. Az F kép M maszkkal végzett vékonyításának egy lépése a következőképp írható le:, ahol, pixelenként értelmezve. Informálisan ez annyit tesz, hogy a maszkot minden pozíción ellenőrizzük, hogy pontosan illeszkedik e a képen lévő pixelekre, és ha igen, akkor az adott pozíción lévő pixelt töröljük. Az illesztést mindig a jelölt, középső pixelen értelmezzük. Ami lényeges, hogy egy iterációban a fenti két maszkkal végezzük el a vékonyítást, majd a maradék 6 db elforgatottal. Az eljárás akkor áll le, amikor az egyik iterációban nincsen változás, azaz egyik maszk sem illeszkedik a 8 közül.



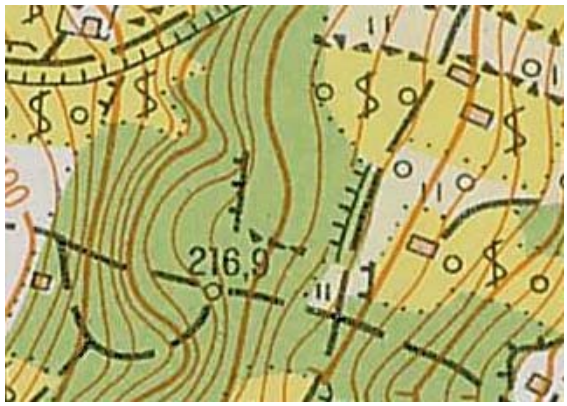
*1.40: Az eredeti kép. 1.41: Az első vékonyítási algoritmus eredménye. 1.42: Morfológiai vékonyítás eredménye. 1.43: Morfológiai vékonyítással egyszerűsítve az 1.41 képet.*

8 Lam L., Lee S. W., Suen C. Y. (1992): Thinning methodologies - a comprehensive survey. IEEE Trans. on Pattern Analysis and Machine Intelligence, Vol. 14, No. 9, pp. 869-887.

## Színek detektálása

Mivel a térképi jelkulcsok azonosításának egyik komponense az, hogy egy terület vagy vonal színét is figyelembe vesszük, ezért nagyon fontos olyan szűrési eljárásokat készíteni, amelyek egy megadott színt felismernek. A színek detektálásának bemenete a kép, kimenete pedig egy bitmaszk, melynek igaz pixeljei jelölik a megadott színű pixelt. Ezt a bitmaszkot aztán többféleképp felhasználhatjuk, például megpróbálhatjuk kicserélni a pixeljeit a mögötte lévő réteg színére. Ez a nagy területek vektorizálásánál hasznos.

Ideális esetben a térképen nincs több 8-10 színnél. Ha a térképek nyomtatása és szkennelése tökéletes lenne, akkor a színek detektálása egyszerű lenne, csak le kéne válogatni az adott színkódú pixeleket. Mivel sajnos ez nem így van, ezért más módszereket kell alkalmazni. A térképi elemek felismerésénél lényeges lépés az, amikor a szkennelt, majd előfeldolgozott kép színértékeit a térképnek megfelelő előre meghatározott színértékekké konvertáljuk. Lényeges információ, hogy a térkép alapszíneit a feldolgozás előtt ismerjük, így jobb eredményt érhetünk, mintha egy




felügyelet nélküli osztályozási folyamatban készítjük a kép pixeleknek a csoportosítását. A szín alapú osztályozás nehézségét az adja, hogy a képen megjelenő színek a nyomtatás és a szkennelés során elég nagy szóródást mutatnak, és a nyomtatás során több szín egymást átfedheti, ami miatt kevert képpontok jelenhetnek meg a képen.

## Színmodellek

A színeket többféleképp is lehet reprezentálni, a következőkben két gyakran használt színmodellt mutatok be.

### RGB színmodell

A legegyszerűbb színmodell, tárolja a szín értékeit csatornánként. Az egyes csatornák értéke mondja meg, hogy a színben mekkora szerepet játszik az adott komponens. Ha minden komponens egyformán erős, akkor egy. szürkeárnyalatot kapunk

**HSL színmodell**

A HSL színmodell egy színt a következő értékekkel ad meg: színezet (Hue), telítettség (saturation), világosság (lightness).

A színezetet úgy adja meg, hogy egy körön 0-359 fokig szerepelnek a különböző színek, 60° a tökéletes piros, 180° a tökéletes zöld, 300° pedig a kék szín.

A telítettség adja meg, hogy mennyire élénk egy szín. Ha ez 0, akkor ez egy szürkeárnyalatot jelen, ilyenkor a színezetnek nincs is szerepe. Az 1-es érték jelenti a legélénkebb, legtelítettebb színt.

A világosságot szintén 0 és 1 közt adjuk meg. Ha ez az érték 0, akkor a szín a fekete, ha 1, akkor fehér.

## Egyszerű színdetektálás

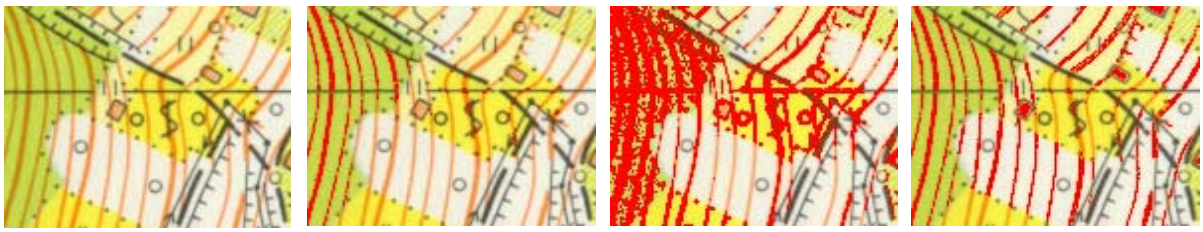
A legegyszerűbb algoritmus, ha definiálunk egy metrikát a pixelek terében, és azokat a pixeleket detektáljuk, amelyek a kiinduló pixel r sugarú környezetébe esnek. Ha a pixeleket RGB színkódjuk alapján vizsgáljuk, akkor ezek a metrikák az $\mathbf{R}^3$-ból ismertek közül választhatók. A legkézenfekvőbb, hogy a pixelek euklideszi távolságát választjuk:

$$\delta(P,Q)=\sqrt{(P[R]-Q[R])^2+(P[G]-Q[G])^2+(P[B]-Q[B])^2}$$

Ha a HSL színmodellel dolgozunk, akkor heurisztikusan más módon is megfoghatjuk a különböző színek közti eltérést. A térképeken detektálandó színeknél fontosabb a színezet, mint a telítettség és az élénkség. Az egyes komponenseket a [0..255] intervallumba transzformálva a következő metrika hasznosnak bizonyult:

$$\delta(P,Q)=\sqrt{\frac{2*(P[H]-Q[H])^2+(P[L]-Q[L])^2+(P[S]-Q[S])^2}{4}}$$

Ezeket a metrikákat megfelelő r környezettel használva elfogadható eredményeket kapunk. Sajnos azonban az ugyanolyan színűnek szánt pixelek nagyon különbözőek tudnak lenni. Például vörös szintvonalak fehér területen inkább rózsaszínek, sárgán narancsszínűek, zöldön pedig barnák. Ezért az egyszerű színdetektálásnak vagy nagy tűréshatárt kell megadni egy lehetőleg jó kiinduló színnel, vagy sok színértéket kell megadni, kicsi tűréshatárral és a kapott maszkokat egyesíteni. Előbbi esetben a detektálás sok hibát fog elkövetni, utóbbi esetben a működés távol áll az automatikustól, gyakorlatilag a felhasználó azonosítja a színeket.



*1.44: Az eredeti kép. 1.45: Egyszerű színdetektálás kis tűréshatárral. Csak egy kis részét találja meg annak, amit szerettünk volna. 1.46: Egyszerű színdetektálás nagy tűréshatárral, láthatóan túl sok hibát ejt. 1.47: Több egyszerű kis tűréshatárú színdetektálás összesítése.*

Más színmodellekkel más metrikák definiálására is lehetőség van, de ezek a fenti problémákat nem oldják meg. Ezért más, erőteljesebb heurisztikákat kellett kitalálni. A következő eljárásokban figyelembe vesszük azt is, hogy az azonos(nak szánt) színek szeretnek egymás közelében lenni, azaz ha már találtunk egy jó pixelt, akkor annak a környezetében lehetünk megengedőbbek.

## Egyszerű színdekompozíció

A feldolgozásnál két színhalmazt használunk fel, az első a térkép területi színei, a második a térkép alakzat színei. Az első szín csoporttal a térképen megjelenő területi információkat jelzik, azaz a különböző felszín borításokat különböző színnel. Míg a második színcsoporttal jelzik az utakat, szintvonalakat, térképi jeleket. Lényeges információ az, hogy a magyar topográfiai térképeken ez a két színhalmaz szinte teljesen elválik, és lényeges a két színcsoport különböző típusú megjelenése a térképeken.

A feladat az, hogy minden képponthoz határozzuk meg azt, hogy melyik területi vagy alakzat színhez tartozik. Ehhez azt a feltevést fogadjuk el, hogy minden térképen megjelenő szín előáll egy területi szín és egy alakzat szín konvex kombinációjaként. Eszerint minimalizáljuk a következő kifejezést:

$$min_{c_t \in C_t} min_{c_j \in C_j} min_{0 \le a \le 1} \left\| \alpha c_t + (1-\alpha) c_j - c_p \right\|_{2,}$$

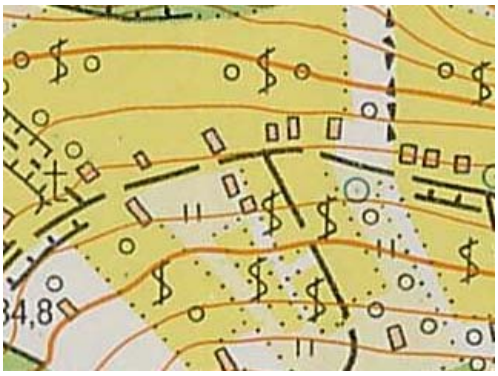
ahol $C_t$ a területi színek halmaza, $C_j$ a alakzat színek halmaza és $c_p$ az aktuális képpont színe. A minimalizálási feladat könnyen megoldható, az összes területi és alakzat színkombinációt kiválasztva meghatározhatjuk a kifejezés minimumpontját. Ezután az optimális eset $c_t$ és $c_j$ értéke közül választható ki a megfelelő szín az $\alpha$ értéke alapján, ha a nagyobb, mint $\alpha_0$ érték, akkor $c_p$-nek megfelelő értéke legyen $c_t$ egyébként, pedig $c_j$.

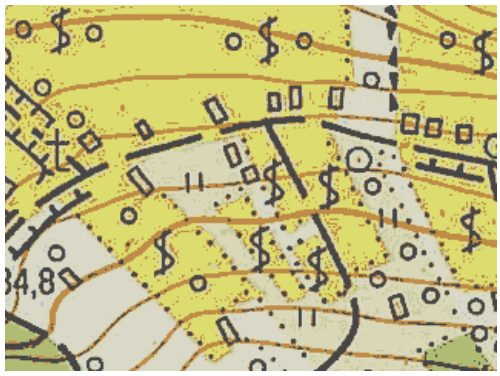
## Környezet alapú színdekompozíció

Az előző módszer hátránya az, hogy érzékeny a színek megválasztására. Ugyanis, ha egy területi szín megegyezik egy másik területi szín és egy alakzat szín konvex

kombinációjával, akkor ez gyakran hibás eredményhez vezethet. Az előző módszert tovább tudjuk úgy fejleszteni, hogy a területi színektől megköveteljük azt, hogy azok a pixel adott környezetében ténylegesen egy területen helyezkedjenek el, és ehhez képest vizsgáljuk a alakzat szín eltérését.

Első lépésben vesszük a pixel egy adott környezetét, és ebben a környezetben legkisebb négyzetes távolság alapján minden pixelt besorolunk egy területi színhez vagy egy alakzat színhez. Ezek után megvizsgáljuk, hogy mely területi színek fordulnak elő domináns mértékben az adott környezetben, azaz 20%-nál nagyobb területen az adott környezetben. Ha nincs ilyen területi szín, akkor a pixel besorolási kategóriáját választjuk eredménynek. Egyébként az egyszerű színdekompozíciónál megismert módszerrel választjuk ki a legjobban illeszkedő eredményt, de már csak a domináns területi színeket felhasználva.





## Javított színfelismerés

Szintén a környezet figyelembevételével szeretnénk eldönteni azt, hogy egy adott pixel megfelelhet-e egy adott területi színre nyomtatott alakzat színnek. Ehhez első lépésként a pixel adott környezetét vizsgáljuk meg, ebben a környezetben többségi osztályozást végzünk legkisebb négyzetes távolság alapján a legnagyobb mértékben előforduló területi színre. Ezek után megvizsgáljuk azt, hogy egy adott alakzat szín a maximális előfordulású területi színnel korrigálva milyen mértékben a pixel színére négyzetes eltérés szerint. Ha minden alakzat szín egy adott küszöbérték feletti távolságot ad a pixel értékektől, akkor nem fogadjuk el alakzat színnek a pixelt, egyébként a minimális távolságú pixelt választjuk. Ebben a modellben nem konvex kombinációnak tekintjük a színkeveredést, hanem additív színkeverési modellt tételezünk fel.

A módszert egy javítási eljárással egészítettük ki. Minden alakzat pixel környezetében még megpróbálunk újabb alakzat pixeleket kiválasztani. Ehhez az adott környezetet minden színkomponens szerint alkalmazzuk a lokális Otsu küszöbölési eljárást, ezáltal nyolc lehetséges kategóriába soroljuk a környezetben lévő képpontokat. Majd azokat a pixeleket, amik 2 pixel távolságon belül vannak a vizsgált alakzat pixelhez és azzal megegyező kategóriába esnek, besoroljuk az alakzat pixelek közé.

## Összetett színdetektáló eljárások

### Bejáráson alapuló színdetektálás

A bejáráson alapuló eljárásban első lépésben egy egyszerű színdetektálást futtatunk a képen valamelyik metrikával. Ezután minden megtalált pixelből elindítunk egy bejárást azok mentén a pixelek mentén, amelyek a megtalált pixeltől nincsenek túl nagy távolságban. Az algoritmus előnye, hogy a műveletigénye nem túl nagy, $O(n)$.

### Otsu küszöbölésen alapuló detektálás

Az Otsu küszöbölésen alapuló algoritmus során az első lépésben egy egyszerű színdetektálást futtatunk a képen. Ezután a kép R, G, B színcsatornáit, mint szürkeárnyalatos képet Otsu eljárásával küszöböljük. Ezek után végigmegyünk a képen és a megtalált pixelek valamekkora környezetében pedig a megadott színűnek választjuk a pixeleket, hogyha legalább 2 csatornán azonos osztályba esnek a kiinduló pixellel. Ezt megtehetjük úgy is, hogy az algoritmust a bejáráson alapuló detektálással kombináljuk, vagyis a kép bejárását olyan pixelek mentén folytatjuk, amelyek legalább két csatornán azonos osztályban vannak az eredeti pixellel.

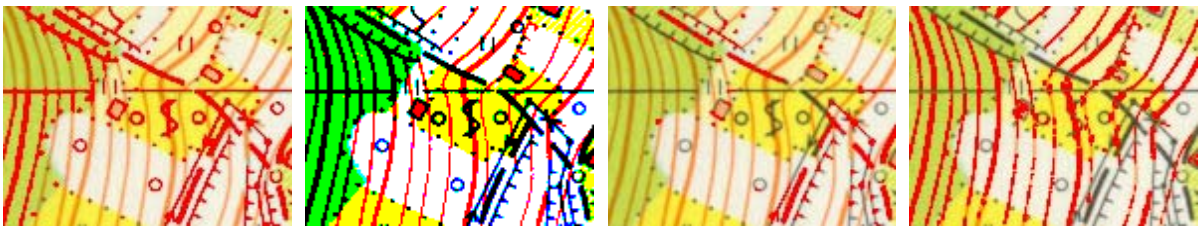
Ennek az algoritmusnak előnye, hogy egy-két színérték megadásával jól lehet azonosítani az ugyanolyan színeket, de műveletigénye $O(n*m*(2k+1)^2)$ nagy konstans szorzóval, ami nagyobb, mint az eddigi eljárásoké.

### A használt színek és egymásra hatásukon alapuló detektálás

A térképek úgy készülnek, hogy először kinyomtatják az alsó térképi rétegeket, a nagy területeket, majd erre rányomtatják a szintvonalakat, térképi jelkulcsot, stb. Körülbelül ki lehet következtetni, hogy a különböző alapszíneken a rányomtatott színek hogyan fognak változni.

Párosítsuk tehát össze az alapszíneket a jelkulcsokat jelölő színekkel. Az alapszínek jellemzően fehér, sárga, drapp, világoszöld, zöld, a rányomtatott színek fekete, barna, világosbarna, narancssárga, piros. Ezeknek a lehetséges színpároknak számoljuk ki a keverését az összeadó színkeverés szabályai szerint, majd ezek után ha valamelyik osztályhoz közeli pixelt találunk, akkor nézzük meg annak környezetét, hogy ott megtaláljuk-e az alapszínt, és ha igen, akkor detektáljuk az adott pixelt.

Ez az eljárás elég erőteljes, hogy ismernünk kell a nyomtatásnál használt színeket. Ez a térképen is meg lehet keresni, de itt nagyon kell figyelni, hogy pl. ne történjen olyan, hogy a zöldre nyomtatott pirosat adjuk meg rányomtatott színnek, hisz az már egy kevert szín. Az algoritmus műveletigénye $O(n*m*(2k+1)^2)$ nagy konstans szorzóval.



*1.48: Elárasztásos detektálás fekete színekre. 1.49: Az összes sávban elvégzett Otsu küszöbölés eredménye. 1.50: Otsu módszerre alapozott fekete-detektálás. 1.51: Otsu módszerére alapozott narancsszín-detektálás.*

## Színek levétele a képről

Amikor felismerünk egy színt, egy természetes cél lehet, hogy eltávolítsuk a képről. A nagy területek vektoritálásánál például a fekete színt legjobb, ha teljesen eltávolítjuk a képről. Ezt megtehetjük úgy, hogy egyszerűen megkeressük a legközelebbi pixelt és azzal kicseréljük az adott színt. Ennek a módszernek hátránya, hogy a háttér változásainál nem feltétlenül a legközelebbi pixel a legjobb választás. Így ezeknél a határoknál a területek széle elromolhat, hullámossá válhat. Ennél egy jobb módszer a kapott színmaszkokat már a szűrések során felhasználni.

# Speciális szűrési eljárások

## Maszkolt szűrők

A maszkolt szűrők alapgondolata az, hogy amikor valamilyen természetű dolgokat szeretnénk azonosítani, akkor vannak objektumok, amelyeket figyelmen kívül szeretnénk hagyni. Ezt tegyük úgy, hogy valamilyen módszerrel készítünk egy maszkot a figyelmen kívül hagyandó pixelekről, majd egy tetszőleges szűrési eljárást alkalmazzunk úgy, hogy a maszkon azonosított pixeleket nem használjuk.

A maszkolt szűrők akkor hasznosak például, ha a nagy területek vektorizálásához szeretnénk jó alapanyagot szolgáltatni. Ehhez azokat az objektumokat kell kiszűrni, amik a területekre vannak rányomtatva: jelkulcsok, szintvonalak, rácsok. Ha ezekből készítünk egy egyesített maszkot, akkor mediánszűrő maszkolt változatával ezeket a részleteket jól el tudjuk távolítani.

Az ilyen maszkolt szűrési eljárások hátránya az, hogy a jó kernelméret kiválasztása nehéz. Ha egy területen kevés figyelmen kívül hagyandó pixel van, akkor egy nagy kernel rossz megoldást szolgáltathat, míg sok ilyen pixel mellett kis kernel túl kevés, esetleg 1-2 pixel alapján próbál dönteni.

A maszkolt szűrési eljárások műveletigénye megegyezik az eredeti szűrés műveletigényével, csak konstans szorzót tesz hozzá minden pixelérték kiszámításának költségéhez.

## Adaptív szűrők

A szűrési eljárások eddig mind úgy működtek, hogy az aktuális pixel környezetéből mindig ugyanúgy választották azokat a pixeleket, amelyekből aztán az új értéket minden esetben ugyanúgy számolták. Így a szűrési eljárások olyan korlátaiba ütköztünk, hogy az élek elmosódtak, vagy eltolódtak, a sarkok le lettek kerekítve. Ez azért van, mert egy objektum és hátterének pixeljeit egyformán vettük figyelembe a szűrésnél.

Az adaptivitás azt jelenti, hogy egy pixel új értékének kiszámításakor vegyük figyelembe a kép lokális tulajdonságait és ez alapján változtassunk a szűrés tulajdonságain. Ez a változtatás lehet a pixelek kiválasztásának menete, a választott pixelekből az új pixelérték kiszámításának mechanizmusa.

### Adaptív pixelkiválasztás, él- és sarokmegőrzés

Az alapötlet az, hogy próbáljuk az objektumokat elkülöníteni a háttértől kernel alatt, valamint a hibákat is hagyjuk figyelmen kívül. Eddig a kernel alatti összes pixelt használtuk, most csak bizonyos pixeleket válasszunk ki. A pixelek választására több eljárás is lehetséges.

**Legközelebbi pixelek választása**

A kernel alatt található pixelek közül az i db. legközelebbit, vagyis számoljuk ki a 2k+1x2k+1 db. pixel valamilyen metrika szerinti távolságát a középső pixeltől, majd ebből az i db. legközelebbit válasszuk ki. Ebben az esetben fontos az i érték jó megválasztása. Ha az élmegőrzés a célunk akkor természetesen adódik, hogy használjuk fel az

$$i=(2k+1)*k+(k-1)$$

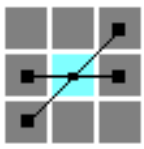
darab legközelebbi pixelt, hiszen így kicsit több, mint a szomszédos pixelek felét használjuk fel. Így ha a vizsgált pixel épp az élen van valamelyik oldalon, akkor nem fogjuk az élt elmosni, mert az új értéket a megfelelő oldalról választott pixelekből fogjuk számolni. Ha sarokmegőrzés a célunk, akkor elég a pixelek körülbelül egynegyedét használni.

Az algoritmus hátránya, hogy műveletigénye nagy, ki kell számolni a kernel alá eső összes pixel távolságát a középponttól, a távolság szerint sorba rendezni és ebből kiválasztani az i darab legkisebbet. A műveletigény így $\boldsymbol{O}(n*m*(2k+1)^2*\log(2k+1))$, de ebben még nincs benne az eredeti szűrés. Másrészt a lokális geometriai elhelyezkedés nincs figyelembe véve, ezért nem biztos, hogy az éleket és sarokpontokat legjobban leíró pontokat választjuk.

**Szimmetrikus kiválasztás**

Mivel az élek úgy jelennek meg a kernel alatt, hogy egy vonal két oldalán különböző színű pixelek vannak, ezért az él szélén lévő pixelek környezetében szimmetrikusan helyezkednek el különböző oldalra eső pixelek. Ezért legyen a kiválasztási eljárás a következő: C legyen a középpontban lévő pixel a P és $P_s$ pedig a C-re szimmetrikusan elhelyezkedő pixelpár. Ekkor válasszuk a P értéket, ha $\delta(P,C)<\delta(P_s,C)$. Ez azért szerencsés választás, mert így a lokális geometriát is figyelembe vesszük.

Ha a sarkok megtartása is célunk, akkor a választás úgy változik meg, hogy a pixeleknek csak a negyedét választjuk ki, méghozzá a következő módon: legyen *C* mint eddig, a középpontban lévő pixel, $P_1$, $P_2$, $P_3$, $P_4$ sorrendben egymás *C* körül 90°-al vett elforgatottjai. Ezek után válasszuk $P_i=argmax_i(\delta(C,P_i))$ pixelt. Így csak a nagyon éles sarkokat fogjuk kicsit lekerekíteni.



*1.52: Szimmetrikus kiválasztás, élmegőrző változat.*
*1.53: Sarokmegőrző változat.*

A szimmetrikus kiválasztás előnye, hogy a lokális geometriai viszonyok figyelembevételével dolgozik, így jobb eredményt ad, mint az i legközelebbi érték kiválasztása. További előnye ehhez az algoritmushoz képest, hogy gyorsabb, mivel itt nem kell rendezni a kernel alatt található pixeleket. A tesztelés összes műveletigénye így $\boldsymbol{O}(n*m*(2k+1)^2)$, de itt még nincs figyelembe véve az eredeti szűrés.

*1.54: Eredeti kép. 1.55: Sarokmegőrző box szűrővel szűrt változat. 1.56-57: 7x7 méretű élmegőrző mediánszűrő első és második futtatásának eredménye. Az 57-es kép már elég jó bemenet a nagy területek vektorizálásához.*

**Adaptív kernelméret**

Az összetett eljárásoknál előfordul, hogy egy kernelpozíció alatt nem találunk elegendő pixelt ahhoz, hogy megalapozott döntést hozhassunk. Ilyen eset például, ha kis kernellel maszkolt szűrést futtatunk és így esetleg 0 db érvényes pixelből próbálunk új intenzitásértéket kiszámolni így olyan pozíción, ahol egyébként egy nagy „igaz" folt fekszik a maszkon. Ha eleve túl nagy kernellel dolgozunk, akkor viszont más helyeken olyan részleteket is eltávolíthatunk, ami nem állt szándékunkban és feleslegesen sok műveletet végzünk.

A megoldást az jelentheti, ha az érvényes pixelek számától függően növeljük a pixelméretet. Az algoritmus más erőteljes szűrők kombinálásával igen robusztus, hátránya, hogy lassú. A műveletigény attól függ, hogy hányszor és mekkorára kell nagyítani a kernelt, valamint, hogy milyen a kiinduló szűrési eljárás.


*1.58: Eredeti kép. 1.59: Élmegőrző maszkolt mediánszűrővel szűrt változat. 1.60: Sarokmegőrző, adaptív méretű maszkolt mediánszűrővel szűrt változat.*

# Kombinált szűrési eljárások a térképek előfeldolgozási lépésben

Most, hogy áttekintettük a képszűrési eljárásokat, amelyek az előfeldolgozási lépésben szóba jöhetnek mint építőkockák, most megnézzük, hogyan lehet őket hatékonyan együtt alkalmazni, hogy a céljainkat elérjük. Mint arról már korábban volt szó, az előfeldolgozás eredménye egyrészt a nagy területek vektorizálásához, másrészt a részletek jó felismeréséhez szükséges.

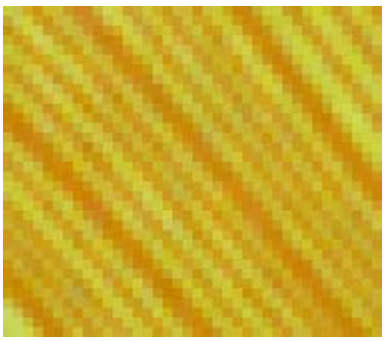
Ezek egyrészt ellentmondó célok, mert a két bemenet nyilván nem lehet ugyanaz. De azt is észre kell venni, hogy a nagy területeket az eredeti térkép jelkulcs, szintvonalak, feliratok és rácsvonalak nélküli változataként kapjuk. Látható, hogy a két cél szorosan összefügg, hiszen ezek szerint el kell különíteni a két térképi réteget.

A következőkben a tapasztalataim szerint jónak bizonyult eljáráskombinációkat és heurisztikákat ismertetem.

## Szintvonalak felismerése

A térképen a nagy területek homogénné változtatásánál az egyik legnagyobb kihívást a szintvonalak jelentik. A szintvonalak egy bizonyos tengerszint feletti magasságot azonosítanak, hogy pontosan milyet, az rájuk van írva. Fontos tulajdonságuk, hogy zárt vonalat alkotnak, azaz önmagukba térnek vissza. A szintvonalak az egész térképen ugyanolyan színnel jelöltek, eltekintve természetesen a már taglalt nyomtatási és színfelismerési hibáktól. Ezen kívül körülbelül azonos vonalvastagság jellemzi őket az egész térképen.

A szintvonalak leszedésére az első ötlet, hogy mivel egy 2k+1x2k+1 méretű mediánszűrő eltünteti a $k$ vastagságú, vagy annál vékonyabb vonalakat ezért alkalmazzuk ezt a stratégiát. Az egyik probléma ezzel a módszerrel, hogy a mediánszűrő lekerekíti a sarkokat és eltolhatja az éleket. Ezt lehetne azzal kezelni, hogy adaptív, szimmetrikus kiválasztású mediánszűrőt alkalmazunk, ekkor azonban még nagyobb kernelre van szükség. Ami azonban ennél is nagyobb probléma, ami egyúttal az adaptív változatot is elveti, az az, hogy bizonyos területeken a térképen olyan sűrűn is lehetnek a szintvonalak, hogy egy mediánszűrő azt a területet a szintvonal színére állítaná. Ezeknek a területek a valóságban nagyon meredek emelkedőket jelentenek, a térképen ezeken a helyeken egymásra torlódnak a szintvonalak.




*1.61: Egymásra torlódott szintvonalak. 1.62: A kép medián szűrt változata nagyon rossz eredményt ad.*

Ez az eljárás azért sem jó, mert az így nem azonosítjuk a szintvonalakat, hanem sok más kis részlettel együtt egyszerűen eltávolítjuk. Jobb lenne, ha a szintvonalfelismerő eljárás eredménye egy, az információt tároló bitmaszk lenne, és azt később maszkolt szűrésekben és a vektorizálás során is használni tudnánk. A szintvonalak felismeréséhez tehát olyan összetett eljárásra van szükség, amely figyelembe veszi azok színét és geometriai jellemzőit is. A következő algoritmus hatékonyan működik:

1) először detektáljuk a szintvonalakat azonosító színt/színeket, kis hibatűréssel
2) ezután az eredeti képen alkalmazzuk a Canny-féle éldetektort, nem-maximális élek kiküszöbölésével és enyhe élküszöléssel.
3) futtassunk a detektált pontokból az eredeti képen speciális szélességi bejárást, ahol azt, hogy lehetséges-e egy szomszéd felé tovább menni, azt két tényező határozza meg:
   - egyrészt, hogy a szomszédos pixel milyen színű, ezt csak enyhe feltételként
   - másrészt viszont a detektált élen való keresztülmenést büntessük erősen
4) A szélességi bejárásban tároljuk a legtávolabbi pontot a kiindulástól. Ha ez a legtávolabbi pont elég messze van, akkor a bejárás során elért összes pontot detektáljuk, mint szintvonalat.

Ezzel az eljárással jók a tapasztalatok, köszönhetően annak, hogy a szintvonalak minden tulajdonságára tekintettel van. Az 1)-3) lépések még csak a szín megtalálását végzi jól, a 4-es lépéssel kerül be az algoritmusba az a jellemző, hogy a szintvonalak „hosszúak”. További előnye az algoritmusnak, hogy számításigénye $O(n*m)$, igaz nagy konstanssal.

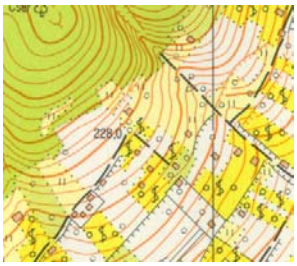
*1.63: Az eredeti kép. 1.64:A felismert szinvonalak.*

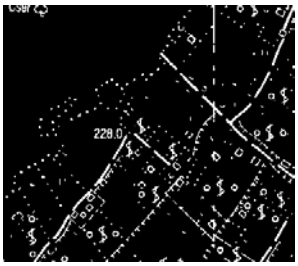
## Lokális Otsu küszöbölés

Az egyéb objektumok - nem szintvonalak - felismerésénél szintén színdetektálásra van szükség. A többi jelkulcs legjellemzőbb színei a barna, fekete és a vizeken sötétkék szín. Ezeknek fontos jellemzője, hogy lokálisan jól elkülönülnek a háttértől. Ezért a következő algoritmussal tudjuk a színek detektálást erősíteni, figyelembe véve ez utóbbi tulajdonságot:

1) Detektáljuk egyszerű detektálással a színeket, kis tűréshatárral.
2) A megtalált pixelek környezetében vágjunk ki a képből egy részt, a méretet úgy határozzuk meg, hogy az ismert jelkulcsok méretéhez hasonló legyen, se sokkal nagyobb, se sokkal kisebb
3) Alkalmazzuk Otsu eljárását minden sávra a képrészleten, majd azokat a pontokat vegyük a színhez, amelyek minden sávon ugyanabba az osztályba tartoznak, valamint bejárással elérhetőek az eredeti pontból

Ez az eljárás azért is jó, mert valamilyen szinten arra is válaszol, hogy melyek az összetartozó pixelek, amik egy jelkulcs előfordulását jelentik. Ennek eredményeképp egy maszkot kapunk.







*1.65: Az eredeti kép. 1.66-67: Lokális Otsu küszöböléssel detektált feketék és barnák. 1.68: Egyesített maszk*

## Többszörösen adaptív maszkolt szűrés

A nagy területek vektorizálásához lényegében le kell szedni a térképről minden jelkulcsot, betűt, vonalat és szintvonalat. Ha a ezeket leválogattuk bitképbe, akkor ezeket egyesítve maszkolt szűrési eljárást alapozhatunk rá. Így biztosan csak azokkal a pixelekkel fogunk foglalkozni, melyek a nagy területekhez tartoznak.

A maszkolt szűrési eljárásnak a él- és sarokmegőrzőnek kell lennie. Ha a sarokmegőrzés szimmetrikus kiválasztás algoritmusát alkalmazzuk, akkor lesznek képrészek, ahol kis kernel esetén nem találunk elég pixelt ahhoz, hogy megalapozott döntést hozzunk. Ezért a szűrőnek méret szerint is adaptívnak kell lennie. Ezek alapján az algoritmus a következőképpen alakul:

1) detektáljuk a szintvonalakat az előbbi algoritmussal
2) detektáljuk a jelkulcsokat jelentő színeket valamelyik algoritmusunkkal
3) egyesítsük a két maszkot pixelenkénti diszjunkcióval
4) minden pixelre k=1-től, *2k+1x2k+1* méretű kernellel indulva
   - számoljuk meg, hogy sarokmegőrző szimmetrikus kiválasztással hány érvényes pixelt találunk az adott kernel alatt
   - ha ez nagyobb, mint egy előre megadott s érték, akkor alkalmazzunk simító szűrőt
   - ha kisebb, akkor növeljük a k-t eggyel

A simító szűrő, amit az algoritmusban alkalmazunk lehet bármelyik, a legjobb, homogén eredményt a mediánszűrővel lehet elérni.


*1.69: Eredeti kép. 1.70: Élmegőrző maszkolt mediánszűrővel szűrt változat. 1.60: Sarokmegőrző, adaptív méretű maszkolt mediánszűrővel szűrt változat.*

## Több út párhuzamos alkalmazása

Az előfeldolgozás lépései nem kell, hogy szigorúan szekvenciális sorrendben kövessék egymást. Jó eredményeket ad, hogyha több különböző eljárást is végigfuttatunk a képen, majd az eredményeket összegezzük valamilyen módon.

A színdetektálásoknál az összetett módszerek mindegyike hatásos, ezért érdemes lehet többet is elvégezni, vagy akár az egyiket használni a másik bemenetének előállítására az egyszerű detektálás helyett.

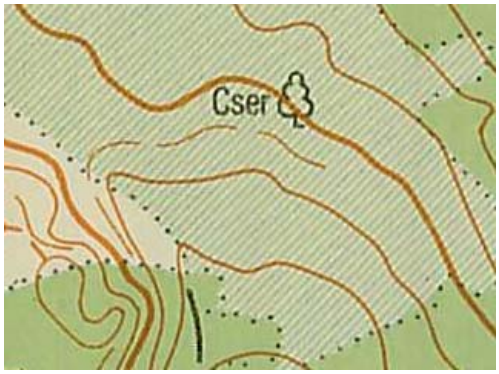
A többszörösen adaptív maszkolt szűrésnél többféle simító eljárás közül is választhatunk. Érdemes összehasonlítani a különféle eredményeket és a nagy eltéréseket újabb vizsgálatoknak alávetni. Hasonlóan hasznos lehet az éldetektálás eredményével összevetni és megnézni, hogy az élek nem romlottak-e el a határoknál.

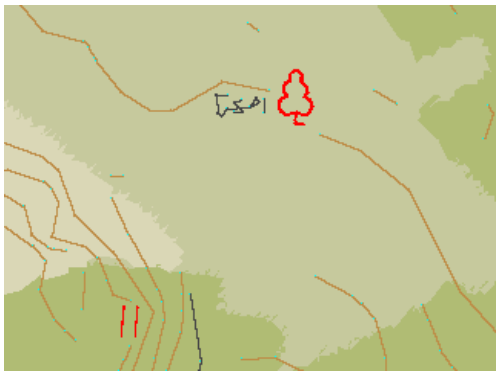
## *Textúrák kezelése*

A textúra felismerés a képelemzés egy nagyon komplex, és jelentős iránya. A térképeken korlátozottan, de előfordulnak nem homogén textúrával fedett területek, ezek felismerésére speciális képfeldolgozási eljárásokat lehet implementálni.

### Egyszerű textúra kezelés

A magyar topográfiai térképen megjelenik egy olyan terület, ami nem homogén textúrát tartalmaz, azaz két eltérő tónusú zöld színnel csíkozott területet. Szerencsére ez a textúra azonban visszavezethető homogén textúrára ugyanis bizonyos szűrési eljárások után közel homogénné tehető. Ezután már alkalmazható tetszőleges osztályozási feladat az előfeldolgozott képen.



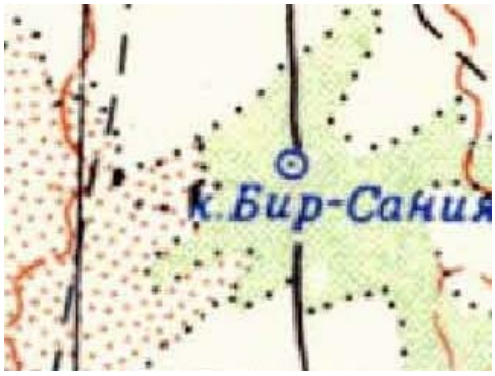
A textúra eltüntetéséhez egymás után több simító szűrést alkalmaztunk. Előfeldolgozásként megjelöltük azokat a pixeleket, amelyek fekete vagy barna pixelekhez tartoznak, azaz alakzat színek. Majd ezen pixeleket, mint maszkot felhasználva egy sarokmegőrző medián szűrőt alkalmaztunk. Majd ezután háromszor alkalmaztunk egy sarokmegtartó box-szűrőt. Ezután a a képen a textúra közel homogén felületnek tűnik. Érdemes megjegyezni, hogy a többszöri szűrés azért szükséges, mert az él- és sarokmegtartó szűrők jellegzetessége, hogy nagy esély van arra, hogy a textúra nem simul ki a képen.

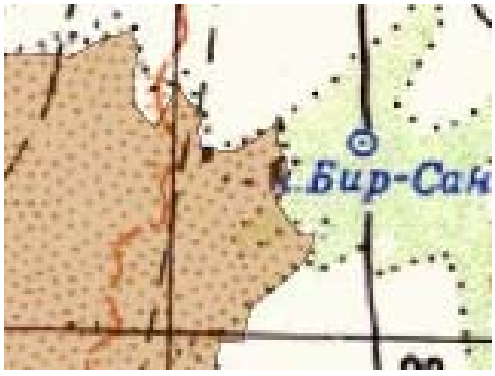
### Pontozott területek felismerése

Speciális textúra felismerési feladatokra külön algoritmus dolgozható, amivel az

adott textúrával rendelkező területek megfelelően felismerhetőek. Ilyen speciális textúra a pontozott terület, amivel egy orosz térkép típus esetében találkoztunk.

A képen a barna pontokat könnyen el tudtuk különíteni egy barna színt kiválasztó szűrővel, majd után a nagy összefüggő területek eltávolíthatjuk a térképről. Ezáltal pontszerű objektumok halmazaként megkaptuk a pontozott területeket. A feladat ezután az, hogy a ponthalmazból meghatározzunk olyan poligonokat, amelyek közel jól fedik le a pontokat. Ehhez első lépésként egy Delaunay háromszögelést állítottunk elő, melyhez Fortune Voronoi diagram készítő algoritmusát használtuk fel. A Delaunay háromszögelés a Voronoi felosztás duális gráfját adja meg, ezáltal a két probléma könnyen átvihető egymásba. A kapott háromszögelésből kitörölvén azokat az éleket amelyek hosszban egy adott küszöbértéket meghaladnak, egy jó közelítését kapjuk azon poligonok éleinek, amelyek leírják a pontozott területeket. Az élekből síkgráfokat építünk fel, majd azon befoglaló lapokat töröljük, amelyek méretben nem érnek el egy adott területet, és azon valós lapokat lyuknak tekintjük egy lapon belül, melyek területe nagyobb egy adott küszöbértéknél.




A módszer előnye az, hogy nem zavarja meg a feldolgozást más térképi objektum. így elég stabil eredményt ad, ellentétben más textúrális jellemzők használatával szemben. Másrészről nagyon gyors feldolgozási módszert biztosít, mivel a Delaunay háromszögelés megvalósítható O(nlog(n)) időben. A módszer más típusú textúrák feldolgozását is lehetővé teszi, ha a textúrából felismerhető olyan sűrű halmaz, ami az egész képen csak a textúrára jellemző.

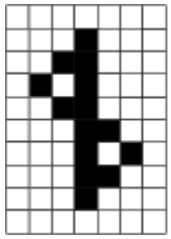
## Mintaillesztések

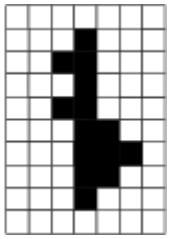
A térképi objektumok előállításához az előre adott térképi jelkulcsokat a feldolgozás során fel kell ismerni, így előállíthatóak a pontszerű objektumok, amik az adott objektumokat jelölik. A mintafelismerést végezhetjük raszteres vagy vektoros adatokon.

### Egyszerű raszteres mintaillesztés

A maszkot ráillesztjük a kép lehetséges területeire, és megnézzük, hogy azonos pozíción különbözik-e a minta és a vizsgált terület színe. Az illeszkedésnek két

költséget határozunk meg. Az első azt mondja meg, hogy a mintabeli pixelek közül hány pixel lett félreosztályozva, míg a második a mintán kívüli pixelek félreosztályozását adja meg. Ha az egyik költség már túl nagy, akkor elvetjük az illeszkedést. Még egy heurisztikát alkalmazunk, ha adott távolságon belül találtunk illeszkedést ugyanerre, vagy egy másik mintára, akkor nem fogadjuk el az illeszkedést.

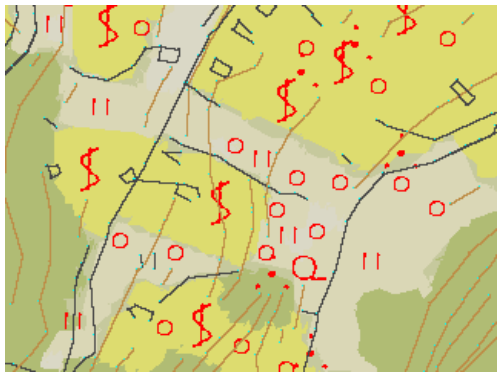




Az algoritmust lényegesen gyorsítottuk. A mintaillesztésnél alkalmazunk egy elő szűrést. Megszámoljuk, azt, hogy a mintában soronként hány minta színű pixel van, és rekurzívan számítjuk azt, hogy ugyanerre mekkora értéket kapunk az aktuális illesztésen. Ezekből az értékekből lehet adni egy alsó becslést a hibára, így nem feltétlenül kell illeszteni a mintát az egész képen.

**Javított raszteres mintaillesztés**

Ebben a módszerben bevezettük a semleges pixelek halmazát. Ezeken a pixeleken nem számolunk illeszkedést, így jobban szabályozható a minta környezete. Az illeszkedés vizsgálat nagyon hasonlóan végrehajtható, mint az egyszerű raszteres illeszkedés vizsgálat esetén. Azonban lényegesen jobb eredményt ez a módszer nem adott, és a fenti gyorsítási módszer hatékonyságát is lényegesen rontotta a semleges pixelek alkalmazása. Ezért gyakorlatban nem alkalmazzuk ezt a módszert.



**Vektoros illeszkedés vizsgálat**

A mintaillesztés másik nagy csoportja a vektoros illeszkedés vizsgálat. Ekkor a minta is vektoros formában, egy poligonként írható le, és az illeszkedésvizsgálat során a képből kinyert poligonokkal vetjük össze.

Erre a feladatra egy dinamikus programozási módszert alkalmaztunk. A módszer

lényege az, hogy a minta poligon határoló vonalát a felismerendő poligon körvonalára úgy próbáljuk áttranszformálni megengedett lépésekkel, hogy az a lehető legkisebb költséggel járjon, költségként pedig a pontok összes mozgatását határoztuk meg. Ha a transzformáció egy adott értéknél kisebb összköltséggel végrehajtható, akkor az illeszkedés elfogadható, egyébként pedig el kell utasítani az illeszkedést. Az algoritmusban kitüntetjük a minta poligon egyik csúcsát, míg fázisonként ezt a pontot próbáljuk megfeleltetni a felismerendő poligon egy adott pontjára. Egy fázison belül adott a két poligonon a kiindulási pont és feltételezzük azt, hogy a két kezdő szakasz iránya megegyezik. Első lépésként a két poligont úgy transzformáltuk, hogy a kezdő pontjuk és kezdő irányuk egybeessen, és a két poligon hossza egyenlő legyen. Az illeszkedés mértékét a következő függvénnyel írjuk le:

$$M(k,l) = min\left( min_{k'<k} M(k',l) + \sum_{i=k'+1}^{k} D_m(i,l), min_{l'<l} M(k,l') + \sum_{i=l'+1}^{l} D_a(k,i), M(k-1,l-1) + d(k,l) \right),$$

ahol $D_m(i, l)$ az i. mintabeli pont távolsága az l. alakzatbeli szakasztól, hasonlóan $D_a(k, i)$ a k. mintabeli szakasztól az i. alakzatbeli ponttól, és d(k, l) a k. mintabeli és az l. alakzatbeli pont távolsága. Az M(K, L) értéket meghatározva megkaptuk az összesített illeszkedést a két poligon között. Az összes fázis közül a minimális illeszkedési költségű változatot választottuk ki.

Ezt a módszert nem tudtuk a valós probléma megoldás során használni, ugyanis viszonylag kis térképi jelek mellett nagyon érzékeny a módszer a hibákra. Az előfeldolgozás során a képen sok esetben a konstruált poligon nagyon eltért az elméleti objektumtól, mivel méretéből adódóan nem volt elég részletes a bitkép, több különálló részre esett szét a poligon, vagy több objektum nem vált el a felismerés során.

**Épületek vektoros illeszkedésvizsgálata**

Speciális minta felismerési feladat az, hogy a térképen az épületeket ismerjük fel. Egy orosz térkép típuson a fekete téglalap alapú területek jelölik az épületeket (igazából az épület tényleges alakját írja le a térkép, de a legtöbb esetben ez egy téglalap). A hagyományos raszter alapú feldolgozás azért nem működik ebben az esetben, mert egyrészt nem egységes a téglalapok iránya, másrészről méretben is nagyon eltérőek lehetnek az épületek.

Az algoritmus első lépésben megkeresi egy fekete pixel kiválasztó szűrővel a lehetséges épület pixeleket. A nagy összefüggő területeket rögtön töröljük a képről, majd a megmaradt összefüggő pixelhalmazok konvex burkát meghatározzuk, és egyben összeszámoljuk azt is, hogy hány darab pixelből áll össze az adott terület. Ezután a forgó érintők módszerével meghatározzuk a legkisebb befoglaló téglalapot, amiről feltételezzük, hogy ez határozza meg az épület helyét a képen.

Két feltételt még vizsgálunk a kapott épületen, ha a pixelek száma nem közelíti meg a téglalap által fedett pixelek számát, akkor nem fogadjuk el az illeszkedést, vagy ha

a téglalap oldalainak aránya nagyobb egy adott küszöbértéknél, akkor szintén elvetjük az illeszkedést.


# 2. Fejezet: Nyersvektorizálás

A topográfiai térképek vektorizálását a bevezetőben három fő lépésre osztottam: előfeldolgozás, nyersvektorizálás, topológiailag helyes geometriai interpretáció. Az előfeldolgozási lépés célja, hogy a nyersvektorizálás eljárásainak kényelmes bemenetet állítson elő. Ezek egyrészt olyan képek, amelyek ránézésre vektoros adatból generált raszteres képnek is tűnhetnek, homogén foltok, jelkulcsok, vonalak, stb. nélkül. Másrészt olyan képekre is szükségünk van, ahol csak a jelkulcsok, vonalak, feliratok vannak rajta egy maszkon.

A nyersvektorizálás feladatai:

- az azonos színnel jelölt területeket osztályokba sorolja
- az összefüggő, ugyanabba az osztályba került foltokból poligonokat állítson elő
- a vonalakat, szaggatott vonalakat vektorizálja
- ismerje fel, hol szerepelnek a jelkulcsok a képen, adja meg azok pozícióját

Észre kell venni, hogy a nyersvektorizálás a topológiailag helyes interpretációval erősen összefügg. A nyersvektorizálás lépései arra irányulnak, hogy a térképi entitásokat nekik legjobban megfelelő természetű geometriai objektumokkal írjuk le. A topológiailag helyes interpretáció lépésének célja pedig az, hogy ezeket az objektumokat valamilyen alkalmas adatszerkezetben tárolja, amely választ ad nekünk olyan kérdésekre, hogy valami mivel szomszédos, hol van, mibe metsz bele, valamint nem enged meg topológiailag helytelen eseteket, pl. a poligonoknak ki kell tölteniük a teljes felületet, nem metszhetik egymást, stb. Ez a lépés nem lineárisan követi a nyersvektorizálást, hanem tulajdonképp azzal párhuzamosan folyik. Mivel azonban a két kérdéskör és az azokhoz szükséges megoldó algoritmusok és adatszerkezetek élesen elkülönülnek, ezért külön tárgyalom őket.

Ebben a fejezetben a nyersvektorizálás feladatainak megoldásához szükséges eljárásokat részletezem, melyek a következők:

- szegmentáló algoritmusok, amelyek a nagy foltokat a színek alapján osztályokba sorolják
- geometriai objektumok, amikre szükségünk van a vektorizálás során
- vonalakat törött vonallá, foltokat poligonná alakító, töröttvonal-egyszerűsítő algoritmusok
- mintafelismerő eljárások

Ezekben az eljárásokban az előfeldolgozás által szolgáltatott képekre koncentrálunk, hiszen a topográfiai térképek jellemzőinek és az előfeldolgozás eredményeinek ismeretében egyes problémák egyszerűbbek, mintha általánosan próbálnánk megoldani őket.

# Pixelek osztályozása

A nagy területek vektrorizálásához először is szükségünk van arra, hogy az előfeldolgozott képen a homogénnek látszó nagy területeket ténylegesen homogénné változtassuk, hogy aztán a szegélyüket megkeresve vektorizálhassuk.

A képelemzés egyik alapproblémája a képek pixeleinek szegmentálása, osztályokba sorolása. A feladat általánosan a következőképp fogalmazható meg: adott célkategóriák esetén egy kép minden pixeljét rendeljük hozzá a célkategóriák egyikéhez a célkategóriák mintáiból vett jellemző adatok segítségével. Ez úgy történik, hogy az adott minták intenzitásai alapján a szegmentáló eljárás egy tanulási folyamat után a kép minden pixeljéről hoz egy döntést, hogy azt melyik kategóriába sorolja. Ehhez felhasználja a pixel intenzitásértékeit az egyes sávokban, a pixel környezetét, stb.

A szegmentálás témaköre nagyon bőséges, a következőkben 3 módszert mutatok be, amelyek illenek a feladathoz és az előfeldolgozott képeken jól működnek.

## Isodata klaszterezéses módszer

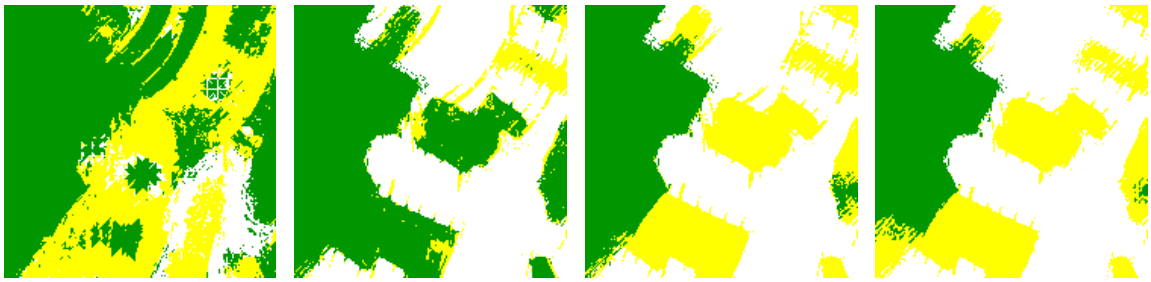
Az Isodata klaszterező eljárás alapelve, hogy iteratív módon k db ideális osztályközéppontot keres a pixelek terében egy d dimenziós ponthalmazban.

Ezt úgy teszi, hogy először k db. különböző kiinduló középpont által meghatározott osztályozást készíti el a pixeljeink közt, majd az egy osztályba került pixelek alapján új középpontokat jelöl ki. Az osztályba sorolást valamilyen metrika segítségével végzi, minden pixelt ahhoz a középponthoz sorol, amelyhez az adott metrika szerint a legközelebb van. Az új osztályközéppont legegyszerűbb meghatározására egyszerű átlagolást is lehet használni, de más heurisztikák is elképzelhetőek, választhatjuk akár a pixelek mediánját is. Ha az osztályközéppontok sokat változnak, akkor az új értékekkel újra elvégezzük az osztályozást. Az algoritmus a következő módon alakul:

1) határozzunk meg $k$ db. osztályközéppontot, legyenek ezek $c_1$, $c_2$, .. $c_k$, legyenek az osztályokba sorolt pixeleket tartalmazó halmazok $H_1$, $H_2$, .. $H_k$
2) minden $p=f(x,y)$ pixelt tegyünk be a $H_i$ halmazba, ahol $i$-t a következőképp kapjuk: $argmin_i(\delta(c_i,p))$ , valamilyen $\delta$ metrikával.
3) számoljuk ki a $c'_i$ új osztályközéppont-értékeket $c'_i=\mu(H_i)$ módon
4) álljunk le, ha a következő feltételek közül valamelyik teljesül
   - $\forall i : \delta(c_i, c_i') < \varepsilon$
   - egy $M$ határt elértünk az iterációk számával
5) ha egyik sem teljesül, akkor $c_i := c'_i$ és folytassuk a 2-es lépéssel

Az eredeti Isodata eljárás klaszterkeresésre is szolgál, így összetettebb, szerepelnek benne heurisztikus lépések a kiinduló osztályközéppontok meghatározására, valamint új osztályok létrehozására. Az előfeldolgozott térkép szegmentálása azonban annyival könnyebb feladat, hogy tudjuk hány osztályunk van, sőt már a kiinduló középpontjaink is elég jók lesznek, így valószínűleg a maximális számnál

kevesebb iterációt is elég végrehajtani.



*2.1-2.4: Az IsoData klaszterkeresésre alapozott szegmentálás 1.,2.,3. és 10. lépésének eredménye egy előfeldolgozott képen, rossz osztályközéppontokból indítva. Ha a térképekkel kapcsolatos ismereteinket kihasználjuk az osztályközéppontok kiválasztásánál, akkor gyorsabban is célt érhetünk.*

Az Isodata eljárás hátránya, hogy egyáltalán nem kerül bele a számításba a térkép lokális geometriája az egyes pixelek környezetében, valamint nagy a műveletigénye: ha a maximális számú iterációt elvégezzük, akkor a műveletigény $O(n*m*k*M)$.

## Szimulált lehűtéses módszer

A szimulált lehűtés[9] módszere valószínűségi alapú meta-algoritmus, mely célja egy feladat optimális megoldásának megoldása véletlenszerű változtatások mentén. Az alapötlet, hogy válasszunk egy teljesen véletlenszerű megoldást, majd minden lépésben válasszunk az aktuális változathoz közeli megoldások közül egyet véletlenszerűen. A közelebbi változatok valószínűsége folyamatosan nő a távolabbiakéhoz képest: úgy, hogy eleinte szinte azonos valószínűséggel választunk az aktuális állapothoz közeli és távoli változatok közül, de később ez a mozgástér egyre kisebb. Ezt úgy érjük el, egy globális t értékkel paraméterezzük a következő állapotok eloszlását.

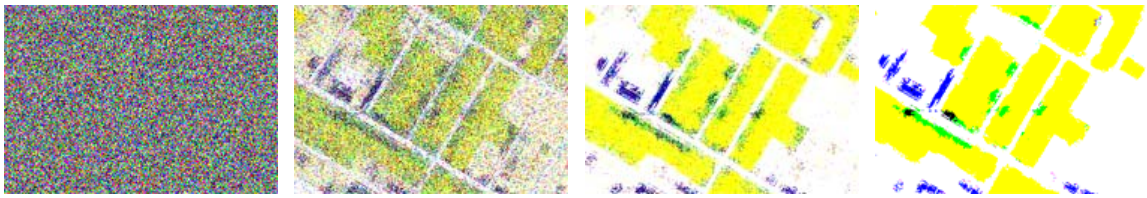
Ezt a gondolatot a következőképp alkalmazhatjuk: kezdetben minden képpont osztályát válasszuk meg véletlenszerűen. Ezután minden lépésben minden pixel következő lépésbeli osztályozását számoljuk ki két komponensből: egyrészt az aktuális pixelérték átszínezésének költségéből másrészt a pixel szomszédaitól való távolságból. Ebből számoljuk ki minden lehetséges osztály valószínűségét és ahol ez a legnagyobb, az legyen a következő állapot. Az algoritmus a következőképp alakul:

1) számoljuk ki minden $c_i$, $c_j$ osztálypárra számoljuk ki a $\delta(c_i,c_j)$ értéket
2) az $s$-edik lépésben minden pixelnek számoljuk ki a következő osztályát az

9 S. Kirkpatrick and C. D. Gelatt and M. P. Vecchi, Optimization by Simulated Annealing, Science, Vol 220, Number 4598, pages 671-680, 1983.

aktuális *c* és a szomszédos pixelek $c_1..c_8$ osztályaiból:

i. a t hűlési paramétert számoljuk ki: $t := \frac{1}{\log(1+(s+1)*0.05)}$
ii. számoljuk ki minden lehetséges $c_k$ osztályra az osztályba sorolás valószínűségét:
   - $p_k = \exp(-\frac{d}{t})$, azaz $-\frac{d}{t}$ paraméterű valószínűségi eloszlással, ahol
   - $d = \delta(c, c_k) + \sum_{i=1}^{8} s_i * \chi(c_i, c_k)$
   - $\chi(ci, ck) = \begin{matrix} 0, ha\ c_i = c_k \\ 1\ különben \end{matrix}$ és $s_i$ az *i*-edik szomszéd súlya, ez a 4-es összefüggésnél 2/12 egyébként 1/12
iii. Válasszuk az osztályok közül azt a $c_{next}$ osztályt, amelyre $p_{next}$ a legkisebb.



*2.5-2.8: A szimulált lehűtés 1., 2., 10. és 50. lépésenként előálló osztályozások egy képrészleten.*

A szimulált lehűtéses módszer előnye, hogy a lokális geometriára is tekintettel van, ugyanis a költségek kiszámolása során a szomszédos pixelektől való távolság is szerepet kap. A hátránya az, hogy lassú, minden egyes iterációban ***O****(n*m*k)* műveletet végez, ahol k az osztályok száma.

## Színdetektálásra alapozott módszer

Az eddigi módszerek eredményei is kielégítőek, annak ellenére, hogy nem használjuk ki az előfeldolgozott térképek speciális tulajdonágait. Egyrészt a térképészeti konvenciók miatt ismerjük a térképen előforduló színeket, valamint a nagy területek tulajdonságait, így a színdetektálásnál tárgyalt módszerek alapelemeit használhatjuk itt is. A célravezető módszer az, hogyha az osztályozást lépésenként végezzük el, folyamatosan eljutva addig, hogy az összes pixelt besoroljuk valamilyen osztályba. Kezdetben egyszerű színdetektálást futtatunk az alapszínekre, így a pixelek egy jó részét besoroljuk egy osztályba. Ezek után a gráfbejáráson alapuló módszerrel vegyük hozzá még azokat a pixeleket is. Azokat a pixeleket, amelyeket még így sem sikerült besorolnunk, azokat az Otsu módszer segítségével azonosítsuk. Így tulajdonképpen egyfajta raszteres „poligonhizlalást" valósítunk meg, ugyanis először csak a kép pixeljeinek egy részét osztályozzuk, aztán folyamatosan veszünk hozzá a detektált színfoltokhoz pixeleket.

*2.9-2.10: A színdetektálásra alapozott szegmentálás kiinduló képe és eredménye.*

Az osztályozások közül ez adja a legjobb eredményt, további előnye, hogy műveletigénye nem túl nagy. Addig, ameddig a lokális küszöbölésre nincs szükség, addig a képpontok számában lineáris, azaz $O(n*m)$, utána pedig attól függ, hogy mennyi pixelt kell besorolni és mekkora ablakmérettel dolgozunk. A legrosszabb esetben a műveletigény $O(n^2*m^2)$, de ez a lehetőség elméleti és azt az esetet írja le, amikor a küszöbölés előtt egyetlen pixelt sikerül csak besorolnunk valamelyik osztályba.

## Hibajavítás

Akármelyik osztályozási eljárást használjuk is, az előfeldolgozás kivédhetetlen hibái miatt érdemes még a tényleges nyersvektorizálás előtt egy egyszerűsítő lépést beiktatunk. A hibák úgy adódnak, hogy a feldolgozás során a színek „kifolyhatnak" az élek és vonalak találkozásánál, egyenetlenségeket okozva.

Mivel a térképek esetén tudjuk, hogy a nagy területek szélén ilyen egyenetlenségek nincsenek, ezért egy egyszerű heurisztikával segíthetünk ezen: osztályonként tegyük meg, hogy a foltokat egy pixellel csökkentjük, majd növeljük. Ezt úgy csináljuk, hogy azokat a pixeleket, amelyek mellett van másmilyen színű pixel, elhagyjuk, majd utána fordítva tesszük ugyanezt. Ez a lépés megváltoztatja a képet, az éleket elsimítja egy kicsit, a pár pixelnyi területeket eltünteti.

# Geometriai objektumok

A vektorizálás építőkövei a geometriai alaptípusok, amelyekkel le tudjuk írni a térképeken talált entitásokat. Ezek az alaptípusok a pont, szakasz, egyenes, töröttvonal, poligon, konvex poligon, téglalap. Ezeket a típusokat és a hozzájuk kapcsolódó műveleteket meg kell valósítanunk, hogy a vektorizálás során kényelmesen tudjuk őket használni. A hozzájuk tartozó műveleteket nem részletezem, csak az összetett geometriai algoritmusokat. A következőkben bemutatom azokat a geometriai objektumokat és algoritmusokat, amelyekre a térképvektorizálás során szükség van.

## Pont

A legegyszerűbb geometriai objektum a pont, melyet természetesen két koordinátájával reprezentálunk. Érdemes meghozni azt a döntést, hogy elkülönítjük a pont és a vektor fogalmát, így a programozás során a fordítóprogram a hibás kifejezések nagyobb százalékát szűri ki. A pont típushoz a szokásos műveletek: elforgatás másik pont körül, távolság ponttól, előjeles távolság egyenestől, projekció egyenesre.

A pont típust használjuk a felismert minták helyének tárolására.

## Vektor

A helyvektor fogalmát leíró leíró geometriai objektum, melyet a ponthoz hasonlóan két koordinátájával írunk le. A szükséges műveletek: hossz, skaláris szorzás, norma, normanégyzet, két vektor által bezárt szög, két vektor által bezárt szög szinusza, koszinusza, három vektor egymáshoz képest vett szögeinek rendezettsége, forgatás origó körül.

## Szakasz

A szakaszt két végpontjával írjuk le. A szakaszokkal egyúttal leírhatnánk az egyeneseket is, de hasonlóan a pontok és vektorok elkülönítéséhez, érdemes itt is külön típusokat létrehozni. A szükséges műveletek: szakasz hossza, szakaszból vektor, szakaszból egyenes, két szakasz metszéspontjának létezése, pont rajta van-e egy szakaszon.

## Egyenes

Az egyeneseket egy pontjukkal és irányvektorukkal írjuk le. A szükséges műveletek: forgatás valamelyik pontja körül, két egyenes metszéspontja, két egyenes által bezárt szög, egy pont az egyenesre esik-e.

## Törött vonal

A törött vonalakat mint pontok sorozatát tároljuk. A szükséges műveletek: teljes

hossz, pontonkénti műveletek, egyszerűsítések

Az egyszerűsítések feladata az, hogy a törött vonal fölösleges pontjait elhagyja. Egy pont feleslegességén általában azt értjük, hogy az azt megelőző és követő pontok által meghatározott egyeneshez túl közel van.

**Távolság alapú egyszerűsítő algoritmus**

A távolság alapú vonalegyszerűsítő algoritmus[10] bemeneteként egy törött vonalat és egy határértéket kapunk. Az algoritmus stratégiája, hogy a törött vonalat rekurzívan két részre osztja a végpontoktól vett legtávolabbi ponttal és a kapott törött vonalrészeket tovább egyszerűsíti. A $p_m$, $p_{m+1}$, .. ,$p_n$ pontok által meghatározott töröt vonalra az algoritmus a következőképp alakul:

1) számoljuk ki az $argmax_i(\delta(p_i, e(p_m, p_n)))$ értéket
2) ha $\delta(p_i, e(p_m, p_n)) > \varepsilon$, akkor folytassuk az algoritmust rekurzívan a $p_m$ .. $p_i$, $p_i$ .. $p_n$ törött vonalakra
3) ha $\delta(p_i, e(p_m, p_n)) < \varepsilon$, akkor állítsuk meg az algoritmust és hagyjuk el a $p_{m+1}$, .. ,$p_{n-1}$ pontokat az egyszerűsített törött vonalból

Mivel az algoritmus a legrosszabb esetben minden lépésben kettéoszt minden töröttvonal-részt, ezért a műveletigény ***O****(n*logm),* ahol az m a kapott egyszerűsített töröttvonal pontjainak száma.

*2.11: Az egyszerűsítés törött vonala. 2.12: az első lépés során kiválasztjuk a legtávolabbi csúcsot, és kettéosztjuk a töröttvonalat. 2.13: a kapott két töröttvonaldarabtól legtávolabbi csúcsok már túl közel vannak, így az algoritmus leáll. Az eredmény a kék töröttvonal*

**Szög alapú egyszerűsítő algoritmus**

A szög alapú egyszerűsítés a távolság alapúhoz nagyon hasonlóan, rekurzívan működik, azzal a különbséggel, hogy azt a pontot választjuk a törött vonalból, amelyből a két végpont a legkisebb szögben látszik, és az így kapott két törött vonal darabra folytatjuk a rekurziót. A két algoritmus műveletigénye megegyezik.

10 D. H. Douglas and T. K. Peucker. Algorithms for the reduction of the number of points required to represent a line or its caricature. The Canadian Cartographer, 10(2):112--122, 1973.

*2.14: Az egyszerűsítés törött vonala. 2.15-2.16: kiválasztjuk azt a csúcsot, amelyből a legkisebb szög alatt látszik a két végpont és kettéosztjuk a töröttvonalat. Itt az egyszerűsítés leáll.*

A törött vonalak nagyon fontosak a vektorizálás során, ugyanis ezt használjuk a vonal típusú objektumok: szintvonalak, utak, stb. leírására.

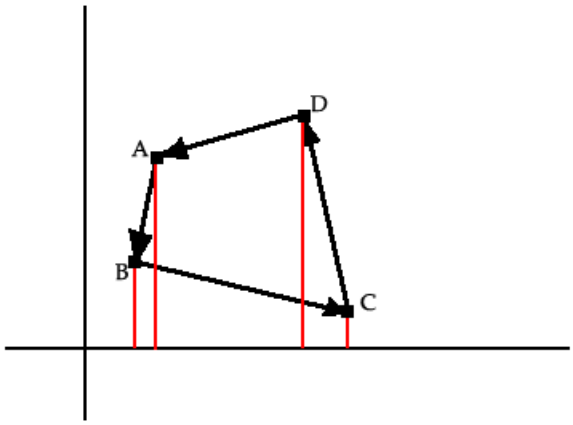
## Poligon

A poligonokat csúcspontjai sorozatával írjuk le. A töröttvonalhoz képest fontos tulajdonsága, hogy élei nem metszhetik egymást. További fontos tulajdonsága, hogy szerepet kap az is, hogy a csúcsok milyen körüljárási irány szerint vannak felsorolva. A térinformatikában elterjedt gyakorlat az óramutató járásával ellentétes felsorolás. Fontos műveletei a területének lekérdezése, irányítás lekérdezése, pontonkénti transzformáció, megfordítása, pont tartalmazásának vizsgálata, konvex burok kiszámítása.

### Poligon előjeles területe, irányítása

Egy poligon irányítását az előjeles területéből lehet megtudni. Ha ez negatív, akkor óramutató járásával egyező, különben ellentétes az irányítás. A poligon területe az élei és az élek x tengelyre vett vetülete által meghatározott trapézok előjeles területének összegeként adódik. Vagyis a $p_1$, $p_2$, .. $p_n$ poligon területét a következőképp adódik:

$$\sum_{i=1}^{n} \frac{(p_{i+1}.x - p_i.x)(p_{i+1}.y + p_i.y)}{2}$$

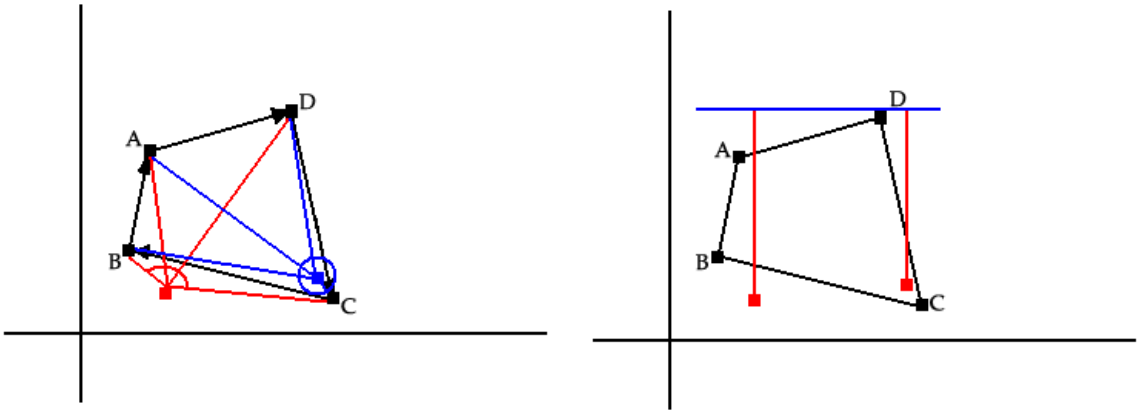



*2.17: Az előjeles terület kiszámítása az előző képlet szerint az AB, BC, CD és DA és x tengelyre vett projekciójuk által meghatározott trapézok területeként adódik*

**Pont tartalmazása**

Azt, hogy egy pont egy poligonon belül van-e, több algoritmussal is eldönthetjük. Az egyik módszer az, hogy összeadjuk az adott pontból az élek, mint irányított szakaszok előjeles látószögét. Ha ezen szögek összege nulla, akkor a poligonon kívüli pontban vagyunk, egyébként pedig poligonon belüli pontban.

A másik módszer az, hogy számon tartjuk a poligon y irány szerinti maximumát, legyen ez $max_y$. Készítsünk egy $p \rightarrow (p_x, max_y)$ szakaszt. Számítsuk ki, hogy a szakasz a poligon hány élét metszi. Ha ez a szám páros, akkor kívül vagyunk a poligonon, különben belül. Amire figyelni kell, ha egy élt úgy metsz ez a szakasz, hogy párhuzamos vele, ilyenkor azt a metszést hagyjuk figyelmen kívül.




*2.18: Az előjeles látószögek összege a piros pontban 0, a kékben pedig negatív. 2.19: A metszéspontok számának vizsgálatával is el tudjuk dönteni, hogy egy pont belül vagy kívül van-e.*

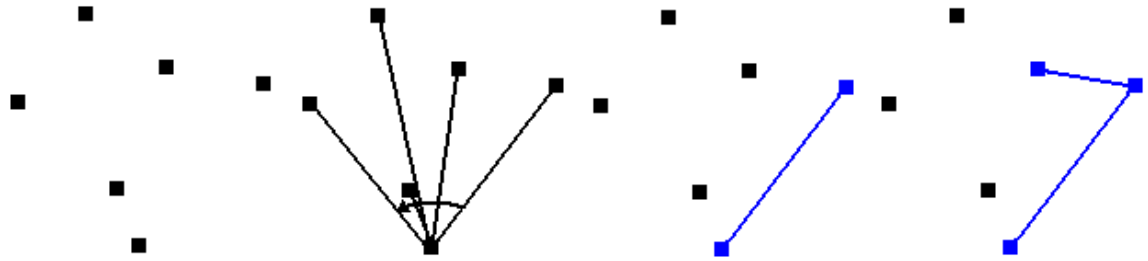
**Konvex burok kiszámítása**

Egy ponthalmaz konvex burkának kiszámítása gyakran előforduló feladat. A térképvektorizálás esetében erre a műveletre a minimális lefedő téglalap megkeresésénél van szükség, ahhoz ugyanis ismerni kell a konvex burkot. A konvex burok kiszámítására több módszer is van, ezek közül kettőt mutatok be.

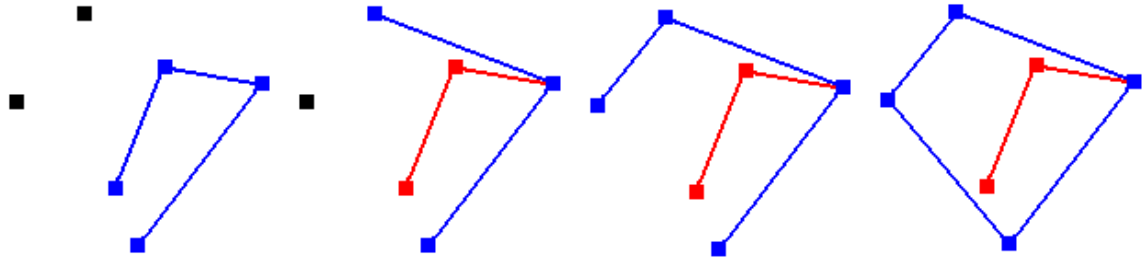
**Graham pásztázás**

A Graham pásztázás[11] módszere először keres egy p extremális pontot, amely biztosan tagja a konvex buroknak. Ezután ekörül a pont körül szög alapján rendezi a többi pontot, végül egy veremalapú művelettel kiszámolja a konvex burkot. Az algoritmus részletesen a p1, p2, .. pn pontok által meghatározott poligonra a következőképp alakul:

1) keressük meg az $argmax_i(p_i.y \in argmin_j(p_j.x))$, pontot, vagyis a legalsó pontok közül a legjobboldalit
2) legyen az e egyenes egy pontja pi és legyen párhuzamos az x tengellyel $\alpha$
3) minden p' ≠ $p_i$ -re számoljuk ki az $\alpha(p_i \rightarrow p', e)$ szöget, és rendezzük p' -ket a szög szerint nagyság szerint növekvő sorba, ez a sorrend legyen $p_{1'}$, $p_{2'}$, .. $p_{(n-1)'}$
4) egy V verembe tegyük bele a $p_i$ és $p_{1'}$ pontokat
5) az 2'..(n-1)' pontokat sorban feldolgozva, a verem tetején lévő pont mindig a $p_t$, az alatta lévő a $p_b$
   - ha a $p_{i'}$ pont a $p_b \rightarrow p_t$ egyenes bal oldalára esik, akkor tegyük be őt a verembe
   - ha pontosan a $p_b \rightarrow p_t$ egyenesre esik, akkor $p_t$-t vegyük ki a veremből, és tegyük a helyére a $p_{i'}$-t
   - ha a $p_b \rightarrow p_t$ egyenes jobb oldalára esik, akkor addig szedjük ki a veremből a pontokat ameddig a verem tetején és az alatt levő pontok által meghatározott egyenes bal oldalára nem esik a $p_{i'}$



11 Graham, R.L. An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set. Information Processing Letters 1, 132-133 1972.



*2.20: A ponthalmaz, aminek a konvex burkát számoljuk. 2.21: A szög szerint rendezett pontok. 22-2.24: Épül a konvex burok, eddig minden lépésben az előző két egyenes bal oldalára esett a következő pont. 2.25: Ebben a lépésben kétszer visszalépünk, ugyanis a következő pont addig a jobb oldalára esik az előző két pont egyenesének. 2.26-2.27: További visszalépések nélkül fejeződik be az algoritmus.*

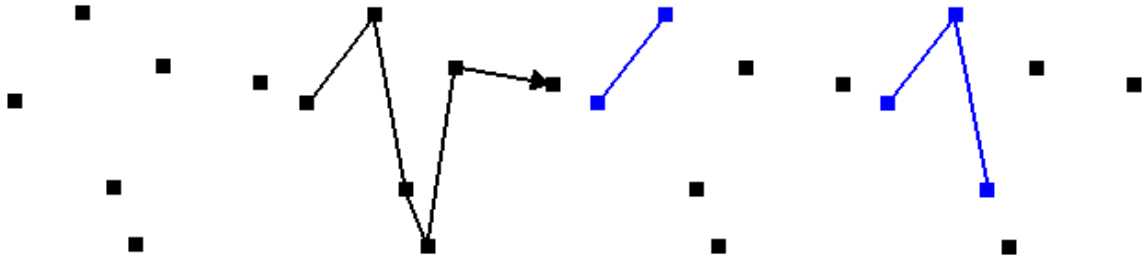
Az algoritmus műveletigénye O(n*logn), ugyanis a rendezést ennyi művelettel tudjuk elvégezni, a burok kiszámítása pedig lineáris idejű.

**Lánc burok**

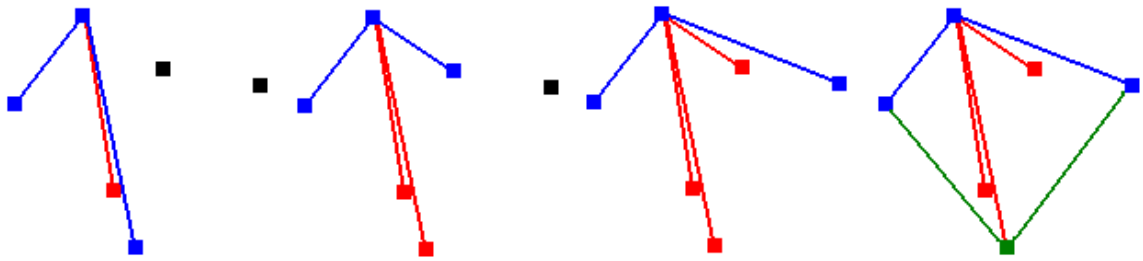
A Graham pásztázás módszerénél szögek szerint kell rendezni a pontokat. Ennél egy egyszerűbb és szebb módszer, ha a pontokat a következő operátor alapján rendezzük:

$$p<q \Leftrightarrow p_x<q_x \vee (p_x=q_x \wedge p_y>q_y)$$

vagyis balról jobbra és fentről lefele. Ezt a sorrendet használjuk fel az előző vermes művelethez hasonlóan egy felső és alsó burkoló építésére. A lánc burok[12] algoritmus műveletigénye megegyezik a Graham pásztázásával, de itt a rendezési összehasonlítás egyszerűbb, így hatékonyabb is valamennyivel.



12 A.M. Andrew, Another Efficient Algorithm for Convex Hulls in Two Dimensions, Info. Proc. Letters 9, 216-219, 1979.



*2.28: A ponthalmaz, aminek a konvex burkát számoljuk. 2.29: A balról jobbra, fentről lefele rendezett pontok. 2.30-2.31: Épül a konvex burok, eddig minden lépésben az előző két egyenes bal oldalára esett a következő pont. 2.32-2.34: Az összes többi lépésben mindig rossz oldalra esik a következő pont, ezért visszalépünk. 2.35: A hasonló módon előálló alsó burokkal egészítjük ki a burkot.*

## Konvex poligon

A konvex poligon olyan poligon, amelyre $\forall p,q \in P, \forall t \in [0,1] : t*p+(1-t)*q \in P.$ A konvex poligon műveletei megegyeznek az egyszerű poligon műveleteivel, azzal a különbséggel, hogy le kell tudnunk ellenőrizni, hogy egy poligon konvex-e, és hogy a konvex poligonoknak könnyen ki tudjuk számítani a minimális lefedő téglalapját.

### Konvexitás ellenőrzése

Egy poligonnál a konvexitás ellenőrzése egyszerű feladat. Sorban meg kell vizsgálni a poligon csúcsait, hogy azok rendre az előző két csúcshoz képest ugyanazon az oldalon vannak-e. Ha igen, akkor konvex, ha nem, akkor pedig konkáv. Ezzel az eljárással tulajdonképpen azt ellenőrizzük, hogy a poligon tartalmaz-e tompaszöget.

### Minimális területű befoglaló téglalap

A minimális lefedő téglalap megkeresésének célja annak a téglalapnak a megkeresése, amely teljes egészében tartalmazza a poligont, és az ilyen téglalapok közt minimális a területe.

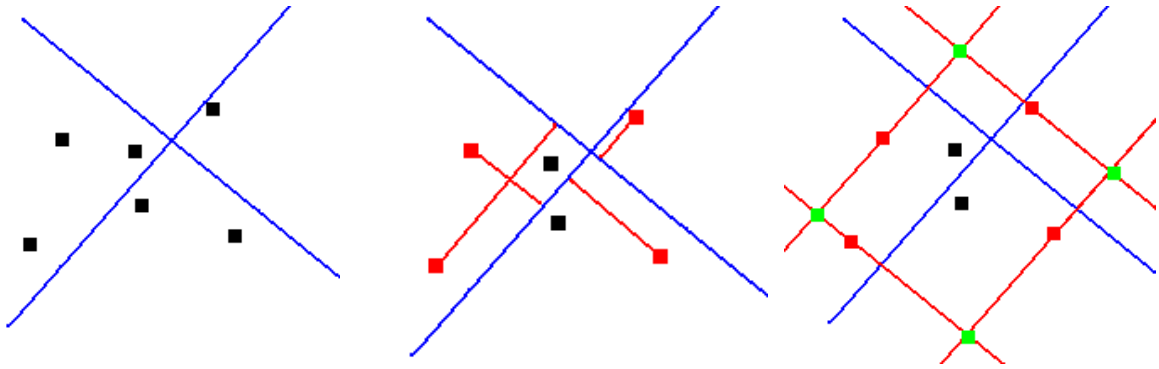
Általánosan nézve, ha van egy ponthalmazunk, és egy egyenesünk, akkor ki tudjuk számolni az azzal az egyenessel egyállású téglalapot, amely lefedi a ponthalmazt. Ezt úgy tudjuk megtenni, hogy meghatározzuk az egyeneshez és a 90°-al vett elfordítottjához legközelebbi és legtávolabbi pontokat.

A minimális lefedő téglalapot tehát nem csak konvex poligonra lehet kiszámítani, hanem tetszőleges ponthalmazra. Figyelembe véve, hogy a minimális lefedő téglalap biztosan illeszkedik a ponthalmaz konvex burkának egyik élére, ezért érdemes csak ezt az esetet tárgyalni. Egyéb ponthalmazoknak és poligonoknak pedig előbb kiszámítani a konvex burkát és utána annak a minimális lefedő téglalapját.

Az illeszkedési tulajdonságból kapunk egy egyszerű algoritmust konvex poligonokra:

1) számítsuk ki a poligon összes éle által meghatározott lefedő téglalapot
2) vegyük ezek közül a legkisebbet

Ez az algoritmus megoldja a feladatot, de műveletigénye $O(n^2)$, ahol n a pontok számát jelenti, ugyanis minden élre meg kell határozni az extremális pontokat. Ez úgy történik, hogy az egyenestől és egy rá állított merőleges egyenestől kiszámoljuk az összes pont előjeles távolságát, és a legkisebb és legnagyobb értékeket megkeressük. Ezek után ezekbe a pontokba eltolva az egyeneseinket megkapjuk az azzal az egyenessel egy állású legkisebb befoglaló téglalapot. Ez $O(n)$ művelet. Ennél a módszernél létezik hatékonyabb is, amiben azt használjuk ki, hogy ha 90°-on belül minden szóba jöhető egyenest kipróbálunk a lefedő téglalap létrehozásához, akkor ezek közt az összes téglalap ott lesz, így elég ezek közül választanunk.



*2.39: A ponthalmaz és az egyenes, amelyikkel egyállású téglalapot keressük. 2.40: Az extremális pontok meghatározása. 2.41: A tengelyek eltolása és a téglalap előállítása ezek metszéspontjaként.*

**Forgó érintők módszere**

A forgó érintők módszere[13] is azt a tulajdonságot használja ki, hogy a minimális lefedő téglalap biztosan illeszkedik a konvex poligon valamelyik oldalára. Készítünk 4 érintőt és addig forgatjuk őket, ameddig körbe ne érünk. Minden forgatás után megnézzük, mekkora téglalapot kaptunk.

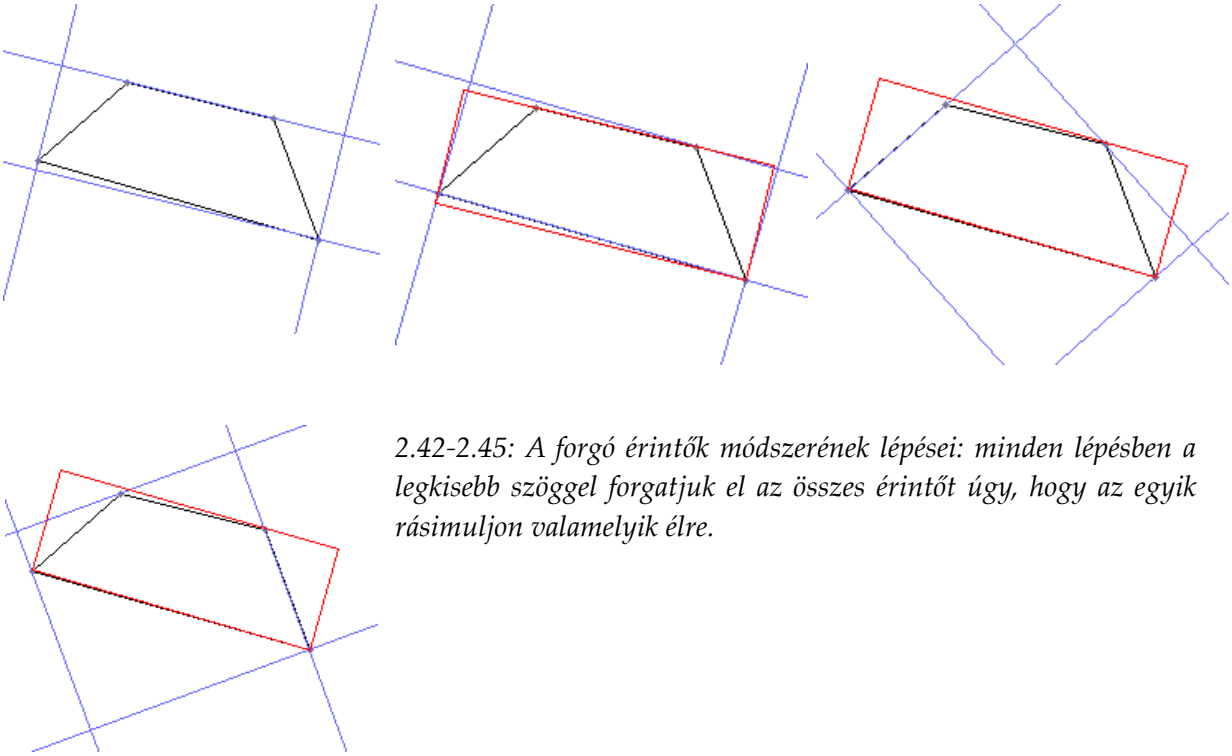
Az algoritmus a következőképp alakul:

1) számoljuk ki a P poligon extremális pontjait az x és y tengely szerint, legyenek ezek $\min P_x$, $\max P_x$, $\min P_y$, $\max P_y$
2) készítsünk egyeneseket, a $\min P_x$ és $\max P_x$ pontokon keresztül az y, a $\min P_y$ és $\max P_y$-on keresztül az x tengellyel párhuzamosokat, ezek legyenek a $c_1$, $c_2$, $c_3$, $c_4$ érintők, az aktuális pontjaik $P_{c1}$, $P_{c2}$, $P_{c3}$, $P_{c4}$, a poligonban rendre következő pontok legyenek $P_{nc1}$, $P_{nc2}$, $P_{nc3}$, $P_{nc4}$
3) $\gamma$-val jelöljük az összes forgatást $\gamma := 0$

13 G. Toussaint, Solving geometric problems with the rotating calipers, Proc. MELECON '83, 1983

4) amíg $\gamma < 90°$

- számoljuk ki a $argmin_i(\alpha(c_i, P_{ci} \rightarrow P_{nci}))$ -vel, hogy melyik érintőt kell a legkevesebbel forgatni, hogy elérjük a következő pontot, ez a szögérték legyen β
- forgassuk el β-val az összes érintőt, ahol kell frissítsük a $P_{ci}$ és $P_{nci}$ pontok értékét, $\gamma := \gamma + \beta$



*2.42-2.45: A forgó érintők módszerének lépései: minden lépésben a legkisebb szöggel forgatjuk el az összes érintőt úgy, hogy az egyik rásimuljon valamelyik élre.*

Ez az algoritmus sokkal jobb, mint az előző, műveletigénye ugyanis $O(n)$, azaz lineáris. Az alapgondolat ugyanaz, attól jobb az algoritmus, hogy megspóroljuk az extremális pontok kiszámítását minden lépésben.

## Téglalap

A téglalap típust a legkisebb befoglaló téglalapok tárolásához érdemes elkülöníteni a konvex poligonokon belül. Műveletei megegyeznek a konvex poligonokéval, természetesen specializálni lehet a terület- és a legkisebb befoglaló téglalap számítást.

## Doboz, befoglaló téglalap

A térinformatikában gyakran előfordul, hogy egy ponthalmaz legkisebb, tengelyállású befoglaló téglalapját (axis-aligned bounding box) kell meghatározni. Ez azt jelenti, hogy olyan befoglaló téglalapot keresünk, amelynek az oldalai az x illetve y tengellyel párhuzamosak. Ez könnyen megtehető, hiszen ha tároljuk a ponthalmaz

minimális és maximális x és y koordinátáit mint $min_x$, $min_y$, $max_x$, $max_y$, akkor ennek a téglalapnak a csúcsai: ($min_x$, $min_y$), ($min_x$,$max_y$), ($max_x$, $max_y$) és ($max_x$, $min_y$). Ezeket az értékeket egy doboz adatszerkezetben érdemes tárolni, amelyben a $min_x$, $min_y$, $max_x$, $max_y$ értékéket karban tartjuk. A doboz műveletei a pont hozzáadása, elvétele, aktuális tengelyállású befoglaló téglalap lekérdezése.

A doboz típus előnye, hogy poligonok (ponthalmazok) metszésének vizsgálatakor olcsó , konstans műveletigényű előellenőrző műveletet kapunk. Ha két poligon befoglaló doboza nem metszi egymást, akkor nem szükséges tovább vizsgálódnunk, mert akkor biztosan nincs metszés, különben el kell végeznünk egy bonyolultabb, teljes ellenőrzést.

### A bevezetett típusok szerepe

A bevezetett típusokra a térképvektorizálás során következő okokból van szükség:

- a térképeken található entitásokat szeretnénk megfeleltetni geometriai objektumoknak: pontoknak, töröttvonalaknak és poligonoknak
- az interpretációhoz szükséges algoritmusok során szerepet kap a többi típus is, melyek a végleges állományban nem, vagy önmagukban nem szerepelnek: vektor, szakasz, ponthalmaz, doboz, téglalap, konvex poligon
- később, a különböző természetű objektumok valamilyen szintű értelmezéséhez szükségünk van azok geometriai összefüggéseire: egy pontról, vonalról vagy poligonról ugyanis bizonyos esetekben a geometriája, környezete és egyéb tulajdonságainak ismeretében további információkra tudunk következtetni

## Raszteres adatok geometriai interpretációja

A legfontosabb cél, hogy a térképeken található dolgokat megfeleltessük a természetükhöz legjobban illő geometriai objektummal. Jellemzően pontokat, töröttvonalakat és poligonokat szeretnénk kapni a végső vektoros állományban, azaz a bevezetett geometriai objektumokkal szeretnénk interpretálni a térképi entitásokat.

Ezt a lépést szokták általános raszter-vektor konverziónak, vagy egyszerűen nyersvektorizálásnak nevezni. A következőkben az alkalmazott alapalgoritmusok bemutatása következik.

### Vonalak vektorizálása

A geometriai interpretáció során a legfontosabb feladat, hogy vonalakat töröttvonalakká alakítsuk, ugyanis a poligonok felépítését is erre a problémára vezetjük majd vissza. Feltételezzük, hogy a vonalrajz egy fekete-fehér bitkép, azaz igaz és hamis értékeket tartalmaz, valamint, hogy a vonalak egy pixel szélesek.

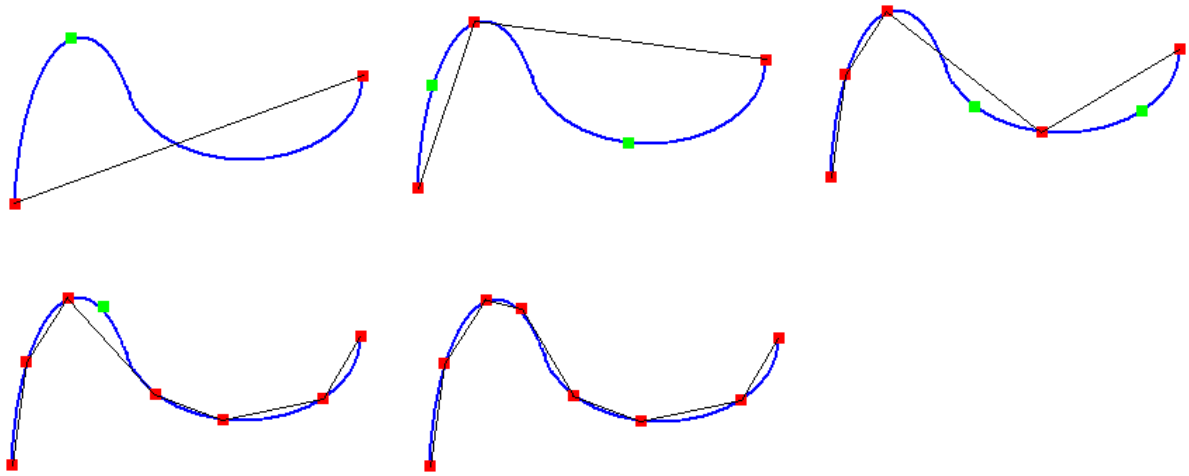
#### Egyszerű vonalak töröttvonallá alakítása

Az legegyszerűbb vonalvektorizálási eset, amikor a vonalban nincs elágazás, vagyis

olyan pixel, amelynek kettőnél több szomszédja lenne a négyszomszédos értelemben. Először ezt oldjuk meg, majd erre vezetjük vissza a bonyolultabb eseteket, az elágazásos és szaggatott vonalakkat.

Az ilyen vonalak vektorizálását a töröttvonalak egyszerűsítéséhez nagyon hasonlóan végezzük. Először megkeressük a vonal egyik végpontját, ez egy olyan pont, amelynek csak egy szomszédja van. Ezután ebből a végpontból a szélességi bejárás stratégiáját követve sorban végigmegyünk a vonal pontjain és ebben a a sorrendben betesszük egy töröttvonalba. Így egy olyan töröttvonalat kapunk, melynek minden éle egy hosszú. Ezt ezután egyszerűsítjük valamelyik töröttvonal-egyszerűsítő algoritmussal.

Fontos az egyszerűsítő algoritmus paramétereinek megválasztása. Egyrészt az íveket meg kell próbálni megtartani, de ugyanakkor a lehető legkevesebb szakasszal leírni az eredeti görbét.



*2.46-2.50: A vonalak vektorizálásának módszere a töröttvonalak egyszerűsítésére alapozva*

**Összetett vonalak töröttvonalakká alakítása**

Az összetett vonalakat, amelyeken vannak elágazások, úgy vektorizáljuk, hogy több egyszerű vonalra bontjuk, majd azokat egyenként feldolgozzuk. Ezt úgy tesszük, hogy azokat a pixeleket, amelyeknek kettőnél több szomszédja van, kivesszük a képből. Attól függően, hogyan kaptuk a vonalas képet, elképzelhető, hogy más pixeleket is ki kell szedni.

A tárolásnál figyelembe kell venni, hogy a töröttvonalak egy töröttvonal részeiként álltak elő és ennek jelentkeznie kell az adatszerkezetben.

**Szaggatott vonalak töröttvonallá alakítása**

Szaggatott vonalak gyakran előfordulnak a térképeken. Ezeket az eddigi eljárások eredményeképp több rövid töröttvonalként kapjuk meg. Természetes, hogy nem így szeretnénk kezelni őket, hanem az eredeti jelentés szerint egy vonalként, amelynek attribútumaként tárolhatjuk, hogy az szaggatott.

Ezt megtehetjük úgy, hogy miután vektorizáltuk a bitképünkön található összes vonalat, utána azokat a töröttvonalakat, amelyek összhossza egy határnál kisebb, megpróbáljuk összekötni más, ugyanilyen vonalakkal. Ehhez csak az ilyen töröttvonalak elején és végén lévő szakaszok meredekségét meghatározzuk. Jellemzően ezek a töröttvonalak egyetlen szakaszból állnak. Ezután az összes ilyen végpont egy környezetében olyan másik végpontot keresünk, amelyből kiinduló szakasz meredeksége körülbelül megegyezik az eredetiével. Ezek közé a végpontok közé egy élet teszünk és a két töröttvonalat egyesítjük.

Ez a megoldás azonban még nem tökéletes, ugyanis olyan eseteket is szaggatott vonalnak detektál, amikor a két vonal ugyan párhuzamos, de a két végpont nem esik egy egyenesbe. Ezért javítsunk az ellenőrzésen úgy, hogy a néhány pixeles távolságban lévő töröttvonalak, $t_1$ és $t_2$ utolsó szakaszainak meredekségét és a végpontokat összekötő szakasz meredekségét vetjük össze és ha az ezek által bezárt szögek abszolút értékének összege nem halad meg egy határt, akkor egyesítsük $t_1$ és $t_2$-t egy töröttvonallá.

*2.51: Az első tesztelési eljárás hibáját mutatja ez a kép, itt egymáshoz közel van két olyan töröttvonalvég, hogy ugyanolyan állásúak, mégsem szeretnénk szaggatott vonalként azonosítani őket. 2.52: A második ellenőrzési módszer helyes eredményt ad.*

Miután ezzel az eljárással végeztünk, érdemes egyszerűsíteni az eredményt, hiszen az eredetileg egyenes vonalakból képezett szaggatott vonalak így sok szakaszként kapjuk meg.

## Foltok poligonná konvertálása

Ez a lépés a vektorizálás során a nagy területek térképi rétegének geometriai interpretációja során szükséges. A foltok poligonná alakításánál először az

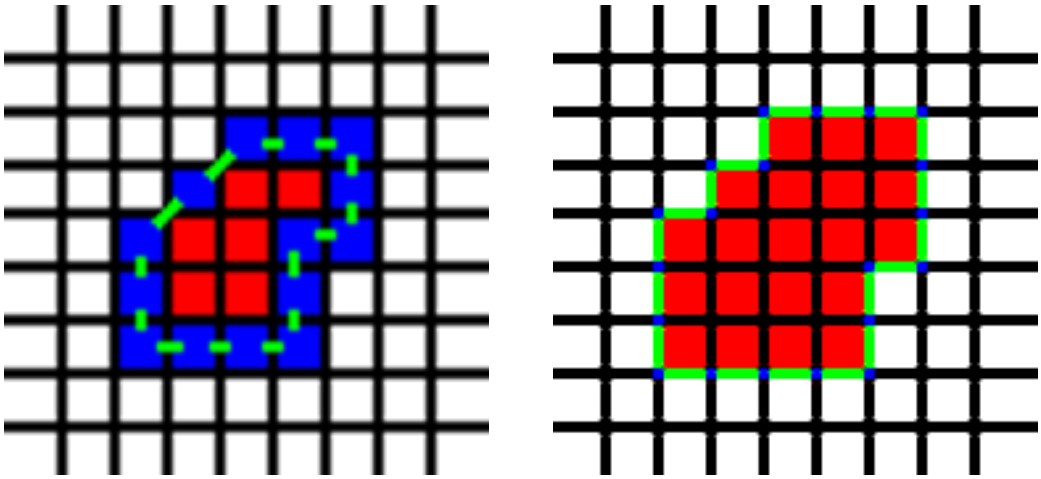
egyszerűség kedvéért feltesszük, hogy a bemenet egy fekete-fehér bitkép, amelyen egyetlen összefüggő folt van. A térképvektorizálás során felesleges valódi bitképet létrehozni, elég az is, hogyha egy folt egy pontját és a szegmentálás utáni osztályát megmondjuk.

**Szegély**

A szegély pontjait kétféleképp is definiálhatjuk.

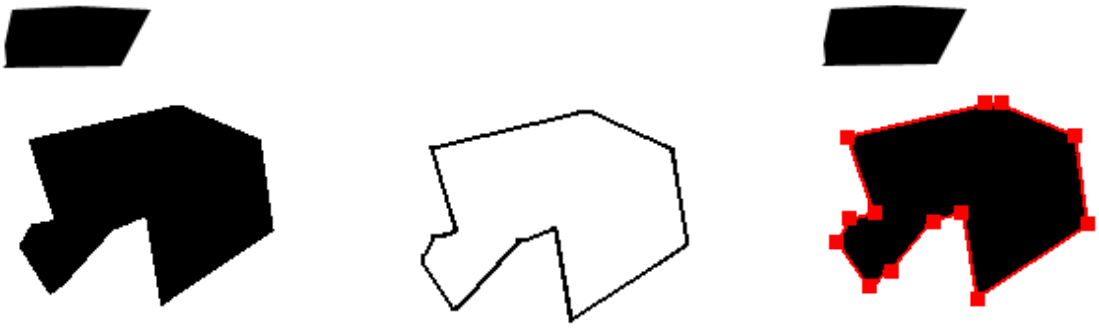
Az egyik megközelítés a szegély pontjai lehetnek azok a pontok a poligonban, amelyeknek van nem-poligon szomszédjuk a képben a 4-szeres szomszédság értelmében. Ez azt jelenti, hogy a poligon szegélyének pixeljeinek a középpontjai lesznek a kiinduló töröttvonal pontjai, és 1 hosszú függőleges és vízszintes, valaminthosszú átlós szakaszok lehetnek benne. Ha csak függőleges és vízszintes szakaszokat szeretnénk, akkor a 8-szoros összefüggőséget válasszuk.

A másik lehetőség, hogy a kép pixeljeit, mint rácsot tekintjük. A szegély pontjai azok a metszéspontok lesznek, melyek egy poligon és egy nem-poligon pixel közös pontjai. Az élek pedig a pixelek oldalai, vagyis a rácspontok közti szakaszok lesznek. Így csak függőleges és vízszintes szakaszaink lesznek a szegélyen.



*2.53: A szélen lévő pixelekként definiált szegély. 2.54: A grid rácspontjaival és éleivel definiált szegély.*

A poligon szegélyét leíró töröttvonalat két lépésben állíthatjuk elő. Először készítsünk el egy raszteres átmeneti állományt, amelyen csak a szegély pixeljei vannak. Ezek után bejárással állítsuk elő a határoló töröttvonalat. Attól függően, hogy melyik szegélyt állítjuk elő, attól függően a rácspontokon, vagy a pixeleken haladjunk. Ezek után egyszerűsítsük a kapott töröttvonalat. Az eredmény lesz a poligon csúcsainak listája.



*2.55: A vektorizálandó poligon. 2.56: A poligon szegélyének maszkja. 2.57: A vektorizálás eredménye, ezt még érdemes lenne egy szög alapú egyszerűsítésnek alávetni.*

## Mintaillesztés

A térképeken a jelkulcsokat, bár van kiterjedésük, geometriailag pontként interpretáljuk. Megtehetnénk, hogy poligonná, vagy töröttvonallá alakítjuk, de a jelkulcsok jelentése leginkább pontszerű. Így a feladat az, hogy ismert mintákat, a jelkulcsokat felismerjük a képen. Ezt a mintaillesztés algoritmusai oldják meg.

A mintaillesztés alapfeladata, hogy egy adott mintának keressük meg az előfordulásait a képen. Formálisan: adott az *f nxm* méretű kép és a *w kxl* méretű minta. Hasonlítsuk össze az f összes lehetséges *(x,y)* pozíciójától számított kxl méretű részképet a w mintával, vagyis hasonlítsuk össze *f(x+i,y+j)*-t *w(i,j)*-vel, ahol *i=1..k, j=1..l*. Természetesen nem szorítkozhatunk csak a tökéletes egyezésekre, hiszen akkor nagyon sok előfordulás, ami nagyon közel van a mintához és ezért szeretnénk, ha felismernénk, kimarad.

### Általános mintaillesztés

Tehát a feladat az, hogy azokat a helyeket fogadjuk el, ahol valamilyen mérték szerint a képen közel vagyunk a mintához, és vessük el, amelyek távol vannak. A feladat tehát a mérték jó meghatározása. Ahogy a színeknél, úgy a minták felismerésénél is többféle mérték elképzelhető. Az egyik lehetséges választás, hogy a minta és a kép aktuális pozíciójának négyzetes eltérésének összegét vegyük:

$$\sum_{i=1,\,j=1}^{i=k,\,j=l} (f(x+i,y+j)-w(i,j))^2$$

Ebben az esetben akkor detektáltuk a mintát, hogyha ez az eltérés nullához közeli. Ez a mérték azonban érzékeny a minta olyan változataira, ahol csak árnyalatbeli eltérés van. Egy sötétebb háttéren egy megfelelően sötétebb mintát nagy eltérésűnek vesz. Ezt korrigálhatjuk úgy, hogy a normalizált négyzetes eltérést vesszük:

$$\sum_{i=1,\,j=1}^{i=k,\,j=l} ([f(x+i,y+j)-\bar{f}(x,y)]-[w(i,j)-\bar{w}])^2 \quad \text{, ahol}$$

$\bar{f}(x,y)$ a kép adott pozíciójában a minta méretében kivágott képrészlet pixeljeinek átlaga, *w* pedig a minta pixeljeinek átlaga. Itt gyakorlatilag az történik, hogy az adott pozíción kivonjuk az átlagot a képből, így a minta előfordulásait ugyanabba a pozícióba toljuk.

**Mintaillesztés bitképeken**

A minta illesztéseinek képtől való eltérésének mérésére sok más módszer is van, nekünk azonban erre nincs szükségünk, ugyanis ezek az intenzitásváltozásokat kezelik, vagy a minták elforgatásainak megtalálására szolgálnak. A térképeken a minták azonban nem szerepelnek átméretezve vagy elforgatva, valamint mivel a jelkulcsoknak a színe is meghatározott, ezért a színdetektálásoknál a szóba jöhető színeket megtalálva és maszkokba téve nincs szükség az intenzitásváltozatok megtalálásának módszereire sem. Így azt az egyszerűbb feladatot kell megoldani, hogy bitképen egy bitmaszkot találjunk meg.

Ezt úgy a legcélszerűbb megtenni, hogy a minta maszkján minden pozícióhoz hozzárendelünk egy különbözési költséget. Ezeket a költségeket úgy választjuk, hogy a teljesen 0 és a teljesen 1 képrészletre is 0,5 legyen a hiba összköltsége. Ezek után az aktuális illesztést az illesztés hibáinak összegeként adjuk meg. A tökéletes illeszkedés költsége nulla. A mintaillesztés műveletigénye ***O****(n*m*k*l)*, ugyanis minden pozíción a mintában található mennyiségű pixelt kell vizsgálni.

Ezen a módszeren gyorsíthatunk kicsit, hogyha megszámoljuk a mintában az egyes sorokban lévő igaz pixelek számát. Ezután a képben minden pozíción soronként megszámoljuk, hány igaz pixel van az egyes sorokban. Ezzel alsó becslést tudunk adni a hibára:

$$\varepsilon > \sum_{i=1}^{i=k} [T_f(x+i,y,y+l) - T_w(i)]^{pozitiv} * d(1) + [T_f(x+i,y,y+l) - T_w(i)]^{negativ} * d(0), \text{ ahol}$$

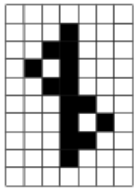
$$x^{pozitiv} = \begin{matrix} x, ha\ x > 0 \\ 0\ különben \end{matrix},$$

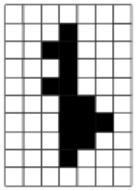
$$x^{negativ} = \begin{matrix} |x|, ha\ x < 0 \\ 0\ különben \end{matrix},$$

$$T_w(i) = \sum_{j=1}^{j=l} \chi(w(i,j)),$$

$$T_f(x,y_1,y_2) = \sum_{j=y_1}^{j=y_2} \chi(f(x,j)).$$

*d(0)* az igaz helyetti hamis, *d(1)* a hamis helyetti igaz értékek költsége. Ezzel tulajdonképp azt írtuk le, hogy ha a mintánál kevesebb igaz érték van egy sorban az illesztésen, akkor akkor annyi pozíción biztosan lesz *d(0)* értékű hibánk és fordítva. Ha ezeket a hibákat összegezzük és az $\varepsilon$ így meghaladja az illeszkedési határértéket, akkor nem kell részletesen ellenőriznünk az adott pozíciót. Az egyes pozíciókon a képen igaz és hamis pixelek számát nem újra és újra számoljuk ki, hanem a ki- és belépő pixelekből és az aktuális értékekből rekurzívan. Így ez minden pozíción ***O****(k)* műveletet vesz igénybe.





*2.58: Az illesztett minta és egy lehetséges előfordulása a jelkulcsnak.*
*2.59: A térképen felismert minták.*

Ha nagyon sokszor megtaláljuk a mintát, vagy az igaz-hamis pixelek eloszlása a képen pont ellenünk dolgozik, akkor ezzel a módszerrel nem gyorsítunk, hanem lassítunk az eljáráson. Ez azonban csak degenerált bemeneteknél van így. A gyakorlatban, mivel kevés illeszkedés van, és sok nagy csupa hamis terület van, így a gyakorlatban az eljárás sokat gyorsul: műveletigénye ***O****(n*m*k)* lesz.

# 3. Fejezet: Topológiailag helyes tárolás

A nyersvektorizálás eredményeként a térképi entitásokat a természetükhöz legjobban illő geometriai objektumok formájában kapjuk meg. Ezek az entitások lehetnek: poligonok, vonalak és pontok. A feladatunk ezek után az, hogy tároljuk el ezeket az objektumokat valamilyen alkalmas adatszerkezetben.

Ennek az adatszerkezetnek a következőkre kell képesnek lennie arra, hogy:

- tárolja a geometriai elemek topológiai viszonyait
- ne engedjen meg hibás eseteket, azaz a poligonok közt ne legyenek rések, azok ne metsszék egymást, stb.
- olyan eseteket is tároljon, amikor egy poligonban egy másik poligon benne van, ugyanis ilyen esetek a térképeken előfordulnak
- tárolja a geometriai entitások attribútumait: egy poligon milyen színű, egy vonal szaggatott-e, stb.
- a tárolt topológiával kapcsolatos kérdésekre könnyen válaszoljon: egy pont hol van, egy törött vonal mit metsz, egy poligon mivel szomszédos, milyen másik poligonokat tartalmaz, stb.

Mivel a geometriai objektumok típusa poligon, töröttvonal és pont lehet. Ezek mind külön térképi réteget írnak le.

A nagy, különböző színű foltokat poligonokkal írjuk le, ezek tartoznak az alsó, területi réteghez. Másrészről a térkép tartalmaz szintvonalakat, utakat, vasútvonalakat, kerítéseket és más vonalszerű objektumokat, melyeket töröttvonalakkal interpretálunk és a hálózati réteg tárol. A harmadik réteg tartalmazza a pontszerű objektumokat, a térképi kulcsokat, amelyet az objektum réteg tárol el.

Természetesen adódik, hogy valamilyen gráf vagy fa adatszerkezetben lenne jó eltárolni ezeket a rétegeket, mert ezekben könnyű leírni a szomszédság és a tartalmazás fogalmát. Minden réteghez konstruáljunk egy gráf típust, amely az azon a rétegen található objektumokat tárolja kényelmesen.

## Területi réteg, síkgráf

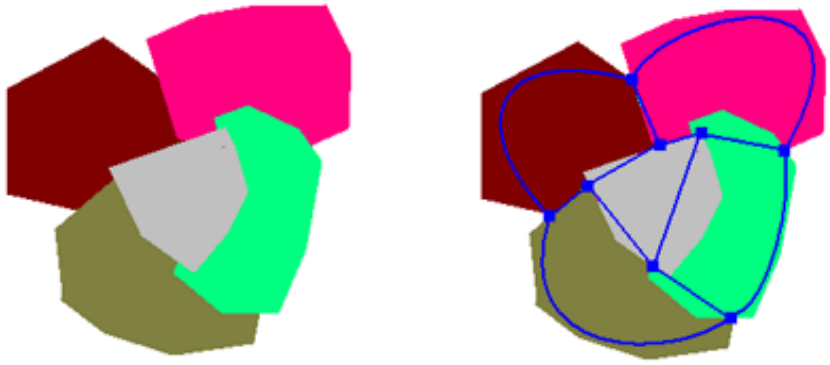
A területi réteget egy speciális összetett gráfstruktúrával ábrázoljuk, amelyet síkgráfnak nevezünk.

Három típusú elemből áll össze: terület, él és elágazási pont. A területek fogják reprezentálni a területei réteg területeit, az élek pedig a területeket elválasztó törött vonalakat, az elágazási pontok a törött vonalak találkozási pontjait.

### Síkgráf

A síkbeli struktúrát valójában két gráfban tároljuk. Az egyik gráf csúcsai az elágazási

pontok, ezek között a pontok élek pedig ott lesznek, ahol ezek között a pontok közt törött töröttvonalak vannak. Ezekben az élekben tároljuk azt is, hogy tőle jobbra és balra milyen területek vannak. Nevezzük ezt a gráfot síkgráfnak.



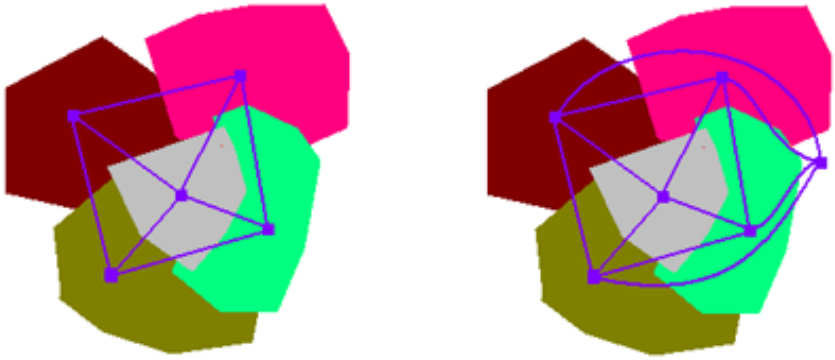
*1: A területek poligonjai. 2: A területek által meghatározott síkgráf.*

**Duális gráf**

A másik gráf a síkgráf duálisa, ebben a csúcsok a területek, az élek pedig azt jelentik, hogy egy területnek egy másik szomszédja. A területhez tároljuk el az őt határoló töröttvonalat is, egyszóval azt a valódi poligont, amit az reprezentál. A duális gráf szerepe, hogy a síkgráffal együtt már leírja a poligonok közti topológiai viszonyokat és azok geometriáját is. Szomszédságra, elérhetőségre vonatkozó kérdésekre így már könnyen válaszolhatunk, de ezzel a struktúrával még nem tudunk egymásba ágyazott területeket tárolni.

**Befoglaló területek**

Definiáljunk ezért a valós területeken kívül egy új típust, a befoglaló területeket. Ezek a valóságban külön poligonként nem megjelenő területek. Legkönnyebb úgy gondolni rájuk, hogy egy befoglaló terület tulajdonképpen egy olyan poligonhalmaz kontúrja, amelynek minden elemét el tudjuk érni az őket leíró síkgráf duális gráfjában. A legkülső befoglaló terület tulajdonképp a poligonstruktúra kontúrja, vagyis legfelső szinten az egész kép, a többi szinten pedig egy teljes lyuk egy valódi poligonon belül.



*3: Az (2) képen lévő síkgráf duális gráfja. 4: A duális gráf a befoglaló területekkel kiegészítve.*

**Befoglalási fa**

A valós és befoglaló területeket hierarchiába rendezhetjük, és fával ábrázolhatjuk. A fa gyökerében a kép kontúrját jelölő befoglaló terület lesz lesz, a gyerekei azok a poligonok, amelyek a duális gráfban elérhetőek belőle. Így minden valós területet hozzárendeltük az őt közvetlenül tartalmazó befoglaló területhez, másrészről az egész képet tartalmazó befogó területen kívül minden befoglaló terület egy valós terület alá tartozik.

A befoglalási fában a gyökérben az egész területstruktúra kontúrja lesz. A páros szinteken befoglaló, a páratlanokon valós területek vannak. Egy befoglaló terület gyerekei azok a valós területek, amelyek belőle a duális gráf bejárásával elérhető.

**A befoglalási fa reprezentációja**

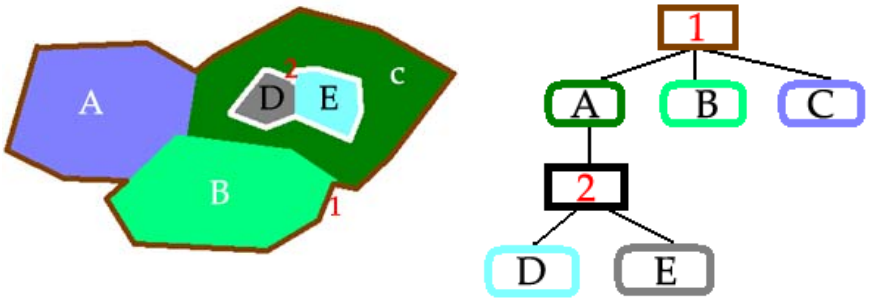
A befoglalási fát nem egy teljesen külön adatszerkezetben tároljuk, hanem a duális gráfot egészítjük ki úgy, hogy az leírja a tartalmazásokat is.

Ha egy poligonon belül van egy beágyazott poligonrendszer, akkor ennek kontúrja lesz a befoglaló területe. A befoglaló területek is megjelennek a duális gráfban: egy befoglaló területnek szomszédja lesz az összes olyan poligon, ami az adott poligonrendszer szélén van. A területekhez tároljuk azt is, hogy befoglaló vagy valós területet írnak-e le.

Ha valós területet írnak le, akkor tartalmazza a határoló irányított éleket negatív körüljárási irány szerint, és a beágyazott befoglaló területek listáját. Ha az adott terület befoglaló terület, akkor tartalmazza a határoló irányított éleket pozitív körüljárási irány szerint és a beágyazott valós területek listáját.

A síkgráfban a szélen lévő éleknek eddig csak az egyik oldalán voltak területek, mostantól a másik szomszédját is beállítjuk, mégpedig a befoglaló területre. Így a síkgráf és duális gráf élei meg fognak egyezni abban az értelemben, hogy a síkgráf éleinek a duális gráf egy másik szerepét adja meg, azaz, hogy az él által reprezentált

töröttvonal két területet választ el.




*5: A valós és befoglaló területek kontúros jelöléssel. A valós területek betűvel, a befoglalók számmal címkézve 6: Az (5) kép struktúrájához tartozó befoglalási fa.*

### A síkgráf és duális gráf élei

Mivel az élek megegyeznek a síkgráfban és duális gráfjában, ezért azokat nem is tároljuk kétszer. A duális gráfba azzal teszünk be egy élt, hogy a bal és jobbolalára eső területeket beállítjuk. Ez azért szerencsés, mert könnyű a használat során egyikről másikra áttérni, valamint egy töröttvonal-változás egyszerre jelenik meg mindenütt, így ilyen változtatással nem képzelhetőek el topológiailag érvénytelen állapotok.

A síkgráf élek irányítatlanok, de egy élben a töröttvonalat mindkét irányítás szerint eltároljuk, hogy az építés során tudunk hivatkozni mindkét irányításra. A reprezentációban a pontokhoz tároljuk a pont koordinátáját, és a kimenő irányított éleket pozitív körüljárás szerint. Az élek irányítatlan éleket írnak le, de hozzárendelünk egy tetszőleges irányítást, ami szerint leírjuk a kezdő és vég pont azonosítóját, és a bal és jobb terület azonosítóját. Ha nincs jobb vagy bal terület, akkor a befoglaló terület azonosítóját használjuk. Ezen kívül letároljuk az élhez tartozó törött vonalat is.

## Előállítás

Az eddig leírt adatszerkezet előnye, hogy támogat minden számunkra fontos műveletet, megőrzi a topológiai helyességet. Ezek után egy olyan eljárást kell adni, amely helyesen felépít egy ilyen struktúrát. A következőkben bemutatom, milyen lépéseket kell elvégezni ahhoz, hogy egy töröttvonal halmazból előállítsuk a síkgráfot, a duális gráfot és a befoglaló és valós területek hierarchiáját. Az eljárás bemenete egy olyan töröttvonal halmaz, amely helyes topológiát ír le és a poligonokat határoló élek csak egyszer szerepelnek benne.

### Síkgráf előállítása

A síkgráf előállításának a bemenete egy helyes töröttvonal halmaz, amelyek a

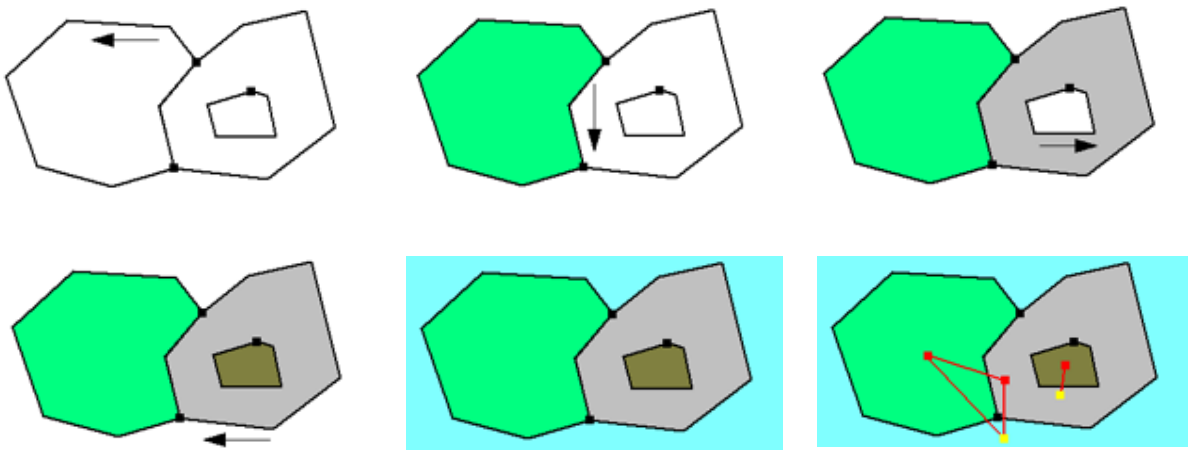
poligonok topológiáját meghatározzák.

1) kigyűjtjük a törött vonalak végpontjait, ezekből létrehozzuk a gráf pontjait és eltároljuk, hogy a síkon melyik pontot jelölik.
2) az összes törött vonalat egy éllel reprezentáljuk
3) az él két végpontjához létrehozott pontok éllistáiba beszúrjuk az élt, figyelve arra, hogy az élek sorrendjének rendezettsége megmaradjon

**Duális gráf előállítása**

A síkgráf előállítása után a duális gráf előállítása következik. Ehhez először az élek feldolgozásával előállítjuk a területeket. Ehhez dolgozzuk fel az éleket valamilyen $e_1$, $e_2$ … $e_{|E|}$ sorrendben. Az $i$-edik lépésben a következőket tesszük:

1) nézzük meg, hogy az $e_i$ mindkét oldalán van-e már terület
2) ha igen, akkor menjünk tovább
3) ha nem, akkor:
   i. hozzunk létre egy új $T$ területet az $e_i$ megfelelő oldalára
   ii. járjuk körbe a területet úgy, hogy az $e_i$ a körüljárás első éle olyan irányítás szerint, hogy T az $e_i$ bal oldalára esik
   iii. a pontokban mindig azon az élen menjünk tovább, ami a körüljárási irány szerint rendezett éllistában a azután következik, amin bejöttünk
   iv. a bejárt élek mindegyikének bal oldalára állítsuk be a $T$ területet
   v. a $T$-hez tároljuk az így kapott körbejárás szerint azt a töröttvonalat, ami határolja, tulajdonképp ez lesz a poligon, amit a $T$ jelöl



*7: A kiinduló töröttvonal halmaz, első lépés iránya. 8, 9, 10: A síkgráf létrehozásának lépései. 11: a végső síkgráf és a duális gráf területei. 12: Duális gráf csúcsai és élei, pirossal a valós területeket jelölő csúcsok, sárgával a befoglalók*

**Befoglaló és valós területek**

A területek felépítésének algoritmusának eredményeképp nem csak a területek és az

azok közti szomszédságok alakultak ki, hanem a poligonok is, amelyek a területet határolják. A körüljárási irányból kiderül, hogy valós, vagy befoglaló területről van szó. Ha ennek a poligonnak a csúcsai helyes körüljárás szerint tárolódnak a határoló poligonban, akkor a terület valós terület, különben határoló. Ezt a poligon előjeles területének kiszámításával tudjuk ellenőrizni.

Ha az összes befoglaló területet megkaptuk, akkor nincs más dolgunk, mint a befoglalási fát megkonstruálni. Ezt a következőképp tehetjük:

1) rendezzük a befoglaló területeket területük szerint csökkenő sorrendbe
2) az első befoglaló terület, azaz a teljes kép, kivételével megkeressük a fában azt, hogy egy befoglaló terület melyik valós területbe tartozik, ezek listáját tegyük bele a duális gráfban a területet reprezentáló csúcsba
3) az aktuális befoglaló területből duál gráf bejárással elérhető területeket rendeljük hozzá a befoglaló területhez, ezek listáját tegyük bele a duális gráfban a befoglalási területet reprezentáló csúcsba

A rendezés miatt nem fordulhat elő olyan, hogy egy befoglaló terület tartalmat egy megelőző befoglaló területet, így nem tudjuk elrontani az egymásba ágyazást.

Ennek az adatszerkezetnek sok előnyös tulajdonsága van:

- Mivel semelyik élt nem tároljuk kétszer, ezért a topológiát nem tudjuk elrontani egy töröttvonal egyszerű megváltoztatásával ha átmetszünk vele egy másik töröttvonalat. Ha azonban erre figyelunk, akkor ezen túl egy él változtatása mindkét poligonra kifejti a hatását, melyet elválaszt.
- Az adatszerkezetben könnyű lekérdezni egy poligon szomszédait, hiszen ahhoz csak a duális gráfban kell a poligont reprezentáló terület szomszédait lekérdezni.
- Támogatja az akárhány szinten beágyazott területeket, és a tartalmazást is könnyű vizsgálni a kiegészítő befoglalási fa vizsgálatával.

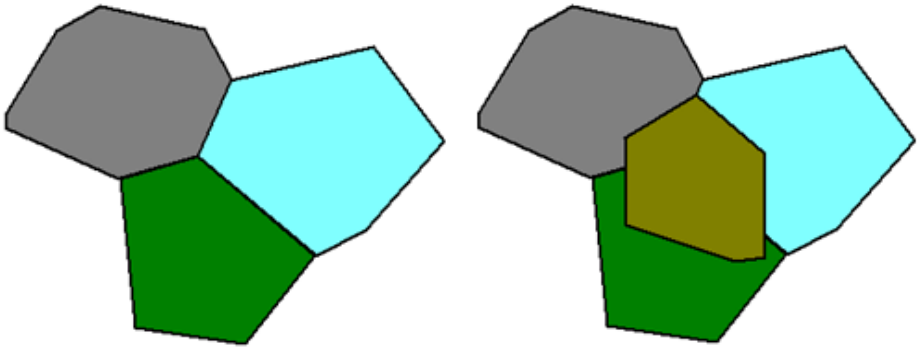
**Síkgráf megváltoztatása**

A felépített adatszerkezettel kapcsolatosan meg kell vizsgálni, hogy a változtatásokat mennyire nehéz elvégezni benne.

Éleket törölni könnyű feladat, ilyenkor azt kell megtenni, hogy megnézzük, milyen területeket határolt el az él. Ha valós területeket, akkor azokat össze kell vonni egy közös területté. Ha valós és befoglaló területet, akkor a valós területet a befoglaló területhez csatoljuk abban az értelemben, hogy a szomszédai innentől azokon a éleken keresztül, amikkel eddig hozzá csatlakoztak, most a befoglaló területhez fognak. Miután ezt megtettük még meg kell nézünk, hogy van-e olyan csúcs, amely csak két él találkozási pontjában van. Ha igen, akkor ezeket kitöröljük, és a két élt, ami a szomszédsági listájában van, összevonjuk.

Az síkgráfok hátránya, hogy a felépített struktúrába új csúcsok beszúrása esetén a duális gráfban többszörös élek lesznek, ezért a csúcsok beszúrását önmagában nem

szabad megengedni, csak hogyha azt élek beszúrása is követi és új területek jönnek létre. Még nagyobb körültekintést igényel új poligonok beszúrása területek közé a lokális topológia megváltoztatásával, mert ez szinte az egész gráf újraépítését követeli.

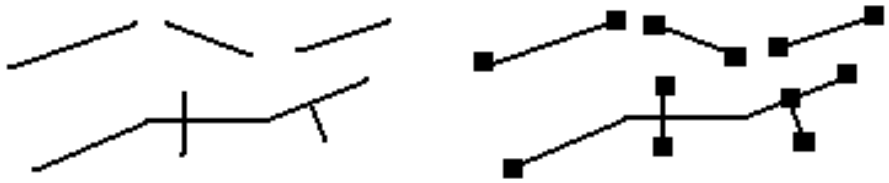


*13-14: Példa nehéz változtatásra, a barna poligon beszúrása a gráfba új csúcsok és élek létrehozásával és törlésével jár, érdemesebb az elejétől felépíteni, mint az eredeti struktúrát változtatni.*

### Hálózati gráf

A hálózati gráf a vonalszerű objektumok tárolására szolgáló adatszerkezet. Konstrukciójához elég a síkgráf adatszerkezet, nincs szükség a duális gráfra és a befoglalási fára, hiszen ebben az esetben területekről nem beszélünk. A gráf csúcsai a töröttvonalak azon pontjai, amelyben elágazás van. Az élek az elágazási pontok közti töröttvonal darabokat reprezentálják. Az élekhez ezen kívül tárolunk attribútumokat, amelyek leírhatják a vonal színét, hogy szaggatott vonal-e és a jelkulcs alapján azt, hogy mit jelentenek: út, szintvonal, stb.

Felépítése a síkgráféhoz nagyon hasonlóan egy töröttvonal-halmaz feldolgozásával történik. A kiinduló töröttvonal halmazra annyi megkötés van, hogy azok ne metsszék egymást. A felépítési algoritmusban a különbség az, hogy az első lépés után megállhatunk, mert ekkor készen vannak a pontok és az élek, a hálózati gráf pedig egyebet nem tartalmaz.



*15: A hálózati réteget leíró töröttvonalak. 16: A (13) kép által meghatározott hálózati gráf.*

### Objektum gráf

Az objektum gráfra azért van szükség, hogy a pontszerű objektumok rétegét is a többi réteghez hasonlóan tudjuk kezelni. Az objektum gráf csak csúcsokat tartalmaz. Felépítése nagyon egyszerű, egy ponthalmaz minden eleméből egy csúcsot kell létrehozni, amelyben eltároljuk a koordinátáit.

## Áttérés más formátumra

Az addig leírt adatszerkezet további előnye, hogy könnyen át lehet térni róla térinformatikai rendszerekkel beolvasható formára. Ez lehet térinformatikai adatbázis, mint az Oracle Spatial, PostGIS, vagy valamelyik szoftver saját fájlformátuma, mint a MapInfo .tab, vagy az ESRE ArcGIS shape[14] formátuma. Mivel ez utóbbi a legelterjedtebb formátum, kvázi ipari szabvány, amelyet az egyes szoftverek egymás közötti adatcserére is használnak, ezért röviden nézzük meg, hogyan lehet a területi-, hálózati- és objektum gráfban tárolt információt shape formátumba menteni.

14 ESRI Shapefile Technical Description - ESRI White Paper, July 1998

### Az ESRI Shape formátum felépítése

A shapefájl geometriai adatokat tárol topológiai információk nélkül, valamint leíró adatokat, amelyek mind egy-egy geometriai elemhez tartoznak. A shape formátum három fájlból áll, egy főfájlból, egy index fájlból és egy dBASE adattáblából. A főfájl egy változó hosszú rekordokat tároló bináris adatfájl, melynek minden rekordja egy geometriai entitást ír le annak pontjainak listájával. Az index fájl tartalmazza az egyes rekordok kezdetének offsetjét a főfájlban. A dBASE fájl attribútumokat tárol a főfájl rekordjaihoz. A tábla minden sora egy ilyen leíró adatot tárol. A főfájl kiterjesztése .shp, az indexfájlé .shx, a dBASE fájlé .dbf.

A shape formátum egy fájlfejlécet és utána rekordok sorozatát tárolja. A rekordokban tárolódnak a tényleges geometriai objektumok. Minden egyes rekord egy fejlécből és egy tartalomból áll. A fejléc mondja meg, hogy hányadik rekordról van szó és hogy a tartalom mező mekkora területet foglal. A tartalom mező a rekord által leírt adat típusát mondja meg (0: NullShape, 1: Pont, 3: töröttvonal, 5: poligon, stb.), majd ezután az objektumhoz tartozó attribútumok következnek. A shape formátum az említetteknél több geometriai típust támogat, de nekünk csak ezekre van szükségünk.

Az indexfájl egy általános információkat tartalmazó fejlécből és rekordokból áll. A k-adik rekord 4-4 bájton ábrázolja, hogy a főfájlban a k-adik rekord melyik bájtnál kezdődik és hány bájt helyet foglal.

A dBASE fájl egy standard .dbf fájl, amelyben az egyes rekordokhoz attribútumokat vagy attribútum-kulcsokat tárolhatunk és összekapcsolhatunk más adattáblákkal.

### Áttérés Shape formátumra

Az ismertetett fájlformátumra áttérni a sík-, hálózati- és objektum gráfról egyszerű, hiszen csak sorban fel kell dolgozni a poligonokat, törött vonalakat és pontokat és az ezeknek megfelelő rekordokat létrehozni a shape fájl pontos specifikációjának megfelelően. Ezeket a rekordokat beszúrjuk a főfájlba és elkészítjük az indexfájlt. Ezzel párhuzamosan a tárolt attribútumokat: poligonok színét, pontok jelkulcsait, vonalak szaggatott voltát, stb. egy dBASE adattáblába egyenként beszúrjuk. Ezzel az adatainkat egy minden fontosabb térinformatikai szoftver által felismert formátumba mentjük.

# 4. fejezet: A térképvektorizálási eljárás

Az eddigi fejezet témái közt érintettük a digitális képelemzés egyszerű és összetett algoritmusait, szegmentálási módszereket, geometriai programokhoz szükséges típusokat és eljárásokat, a nyersvektorizálás alapjait, valamint felvázoltunk egy olyan adatszerkezetet, amely alkalmas arra, hogy térbeli információkat topológiailag helyesen tároljon és amelyben lehetséges a geometriai entitásokhoz attribútumokat rendelni. Ezeket az eljárásokat, típusokat és adatszerkezeteket mind azzal a céllal vezettük be és vizsgáltuk, hogy alkalmasak legyenek egy összetett térképvektorizálási eljárás építőkockáinak.

A következőkben megnézzük, hogy az eddig ismertetett algoritmusok segítségével hogyan lehet a szkennelt papírtérképekből vektoros adatokat kinyerni és végtermékként egy térinformatikai rendszerek által felismert adatformátumot előállítani.

## Konfigurációs beállítások

Az eljárások többször hivatkoztak a jelkulcsok mintáira, az alapterületek, utak, szintvonalak jelölésének ismert színeire. Ezeket a paramétereket az eljárásoknak meg is kell kapniuk, ezt a legegyszerűbb egy konfigurációs fájlban megadni. A konfigurációs fájlból várjuk:

- a szintvonalak színét
- az alapterületek (mező, szántó, vizek) színeiből két pixelnyi mintát
- az utak, vasutak vonalainak színét
- az elképzelhető jelkulcsok maszkjait

A feladat általában nem egy darab térképoldal vektorizálása, hanem nagy mennyiségű, egyszerre, ugyanazzal a jelkulccsal készült térképé. Ezért ezt a rendszerkonfigurálást nyugodtan megkövetelhetjük, hiszen 10-15 szín és ugyanennyi jelkulcs beállítása a kézzel végzett teljes vektorizáláshoz képest is elenyésző munka. Ezt a konfigurálást támogathatjuk grafikus felülettel is úgy, hogy a felhasználó egyszerűen, a színekre kattintva és a jelkulcsokat kijelölve végezhesse el.

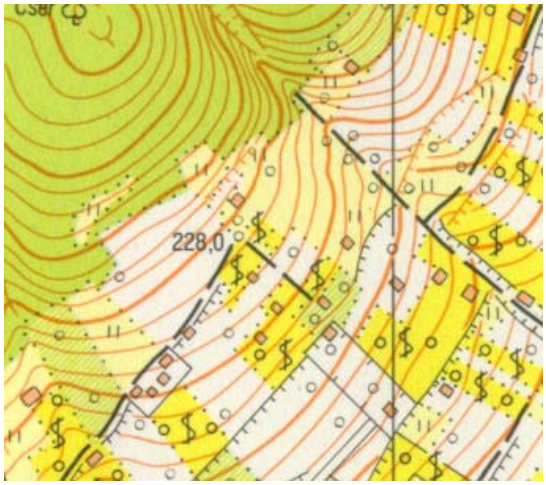
## Előfeldolgozás

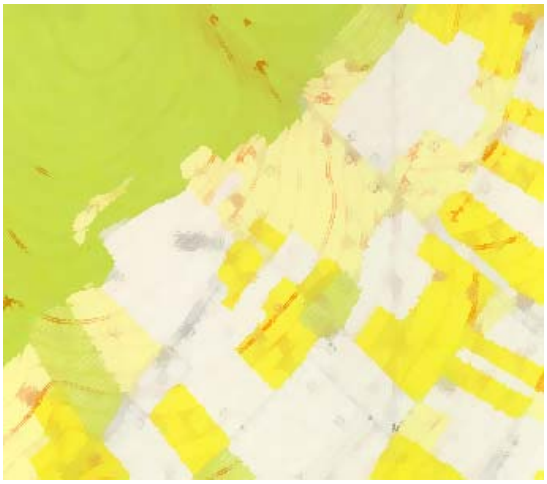
Az előfeldolgozás során a célunk, hogy előállítsunk egy olyan képet, amelyet a területi réteg előállításánál használunk. Ezen kívül elő szeretnénk állítani két bitmaszkot, az egyiken a szintvonalak, a másikon az összes többi, nem területi réteghez tartozó térképi objektumot szeretnénk látni.

Ehhez először is hisztogramkiegyenlítéssel javítjuk az eredeti kép minőségét. Kis méretű, 3x3-as Kuwahara-féle, vagy sarokmegőrző medián szűrővel tovább javítjuk a képet, hogy a „só és bors” típusú hibákat eltüntessük. Az így kapott képet fogjuk a továbbiakban használni a területi réteghez szánt kép előállításához.

## Területi réteg előfeldolgozása

Ehhez a képhez először is előállítjuk azokat a maszkokat, amelyeken a nem területi réteghez tartozó színeket tárolják. Ezek a színek a magyar topográfiai térképeken jellemzően a barna, piros, fekete, de a konfigurációs fájl segítségével más térképekre is felkészülhetünk. A maszk előállítása az előfeldolgozási fejezetben az összetett színdetektálásoknál látható eljárásokkal történhet. A legjobb módszer az egyszerű színdetektálás eredményének lépésenkénti javítása először bejárással, majd lokális Otsu-féle küszöböléssel végül a színek egymásra hatásának vizsgálatával. Ha így előállítottuk a maszkot, akkor adaptív méretű, sarokmegőrző maszkolt medián szűrővel állítjuk elő a nagy területeket tartalmazó képet. Egy jó módszer, ha a szűrő méretének adaptivitását korlátok közé szorítjuk és inkább többször futtatjuk végig a képen, így jobb eredményt kapunk.



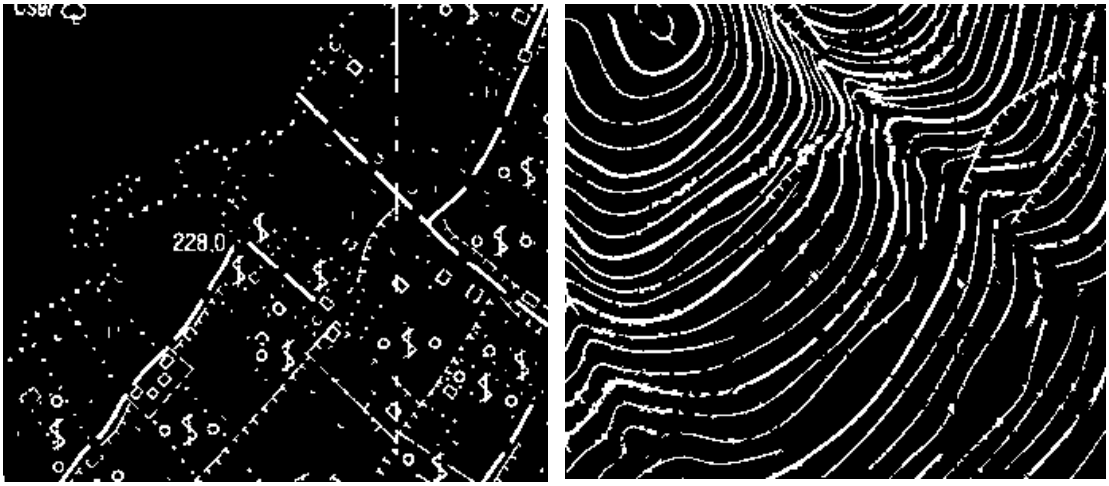
*4.1: Az eredeti kép. 4.2: A nagy területek felismeréséhez előkészített változat.*

## Maszkok előállítása

Az előző lépésben előállítottunk egy maszkot, amelyen azok a pixelek szerepeltek, de előtte már javítottuk a képet egy mediánszűrővel. Ez a kis részleteket nem hagyja helyben, ezért maszkok előállításához az eredeti kép hisztogram-kiegyenlített változatát használjuk. Ezen a képen színdetektálást futtatunk, amely színdetektálás bemenete a jelkulcsok, vonalszerű objektumok színei.

Mivel a szintvonalak színe nem feltétlenül egyedi, ezért a szintvonalak maszkját külön, a szintvonalkereső eljárás segítségével állítjuk elő. Detektáljuk a szintvonalak színeit, így az előző maszk egy részét kapjuk meg, majd a Canny éldetektor eredményét figyelembe véve egy bejárást hajtunk végre minden pontból és így következtetünk arra, hogy szintvonalon vagyunk-e. A szintvonalakat képpontonkénti kizáró vagy művelettel eltávolítjuk az első maszkról.

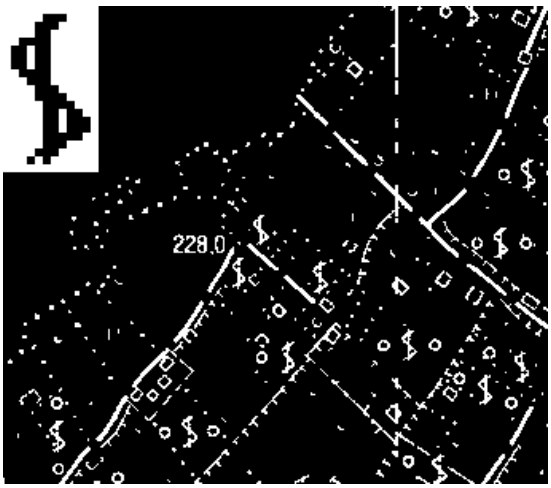
A szintvonalakat azért tároljuk külön is, mert azok a többi vonalszerű objektumtól eltérnek és igazából a területi réteggel kapcsolatos információt hordoznak, valamint később domborzatmodellt hozhatunk létre az adott területről.

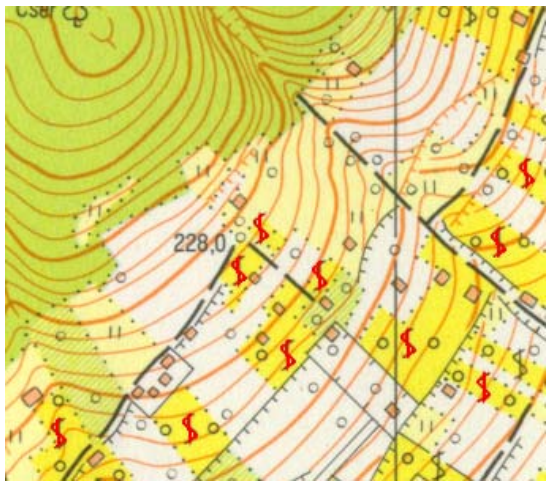



*4.3: A 4.1 kép fekete színeket tartalmazó maszkja. 4.4: a 4.1 kép szintvonalait tartalmazó maszk.*

## Objektum réteg előállítása

Az előfeldolgozott képekből az objektum réteg előállítása a legegyszerűbb. Az ismertetett bitképekre specializált mintaillesztés segítségével megkeressük a maszkon az egyes minták előfordulásait. Egy heurisztikát használunk még: előfordulhat, hogy egy-két pixel eltolással ugyanarra az előfordulásra illesztve a mintánkat, mindig találatot kapunk. Ezért az egymástól maximum 5 pixel távolságra lévő előfordulásokból csak egyet hagyunk meg, így kiküszöböljük a többszörös detektálásokat.




*4.5: A feketéket tartalmazó maszk, amelyen illesztjük a bal felső sarkában látható szőlőjelet.*
*4.6: A felismert előfordulások a képen. Az objektum gráfban az előfordulások csúcsok lesznek.*

A találatokat a síkon egy ponttal reprezentáljuk, majd felépítjük az objektum gráfot. Ami fontos még, hogy a megtalált mintákat leszedjük a maszkról, hogy azokat a hálózati réteg előállításakor ne használjuk fel még egyszer.

## Hálózati réteg előállítása

A hálózati rétegnél két maszkkal dolgozunk. Az egyik a szintvonalak maszkja, a másik pedig az objektum gráf felépítése során redukált maszk. Mindkét maszkon élvékonyító szűrést hajtunk végre, egyesítjük őket, majd ennek eredményét morfológiai élvékonyítással tovább egyszerűsítjük. Ez azért fontos, mert az első eljárás hagyhat csomókat a képen és a morfológiai élvékonyítás ezeket eltünteti. Azért nem érdemes rögtön a morfológiai élvékonyítást alkalmazni, mert az viszont a vonalakon sok kiálló rövid szakaszt hagy. A vékonyított szintvonalakat tartalmazó maszkot megtartjuk, mert ebből fogjuk tudni, hogy egy töröttvonal szintvonal-e.

Az eredményből ezek után töröttvonal halmazt készítünk. Ehhez megkeressük azokat a pontokat a képen, amelyekből egy, vagy kettőnél több felé lehet továbbmenni. Ezt úgy ellenőrizzük, hogy a szomszédos pixeleket végigjárjuk és figyelünk, hányszor léptünk igaz pixelről hamisra. A töröttvonalak előállítása a maszkból a következő eljárással történik:

1) minden pixelt járjunk körbe és nézzük meg, hogy ezen az úton hányszor léptünk igaz pixelről hamisra.
2) ha ez nem kettő, akkor jelöljük meg a pixelt és hozzunk létre hozzá egy csúcsot
3) minden pixelre, aminek van csúcs szomszédja nézzük meg a többi igaz szomszédját
   - ha van ezek közt, ami nem csúcs, akkor abba az irányba lépkedünk amíg nem találunk egy csúcs pixelt, és ezekből létrehozunk egy törött vonalat, az

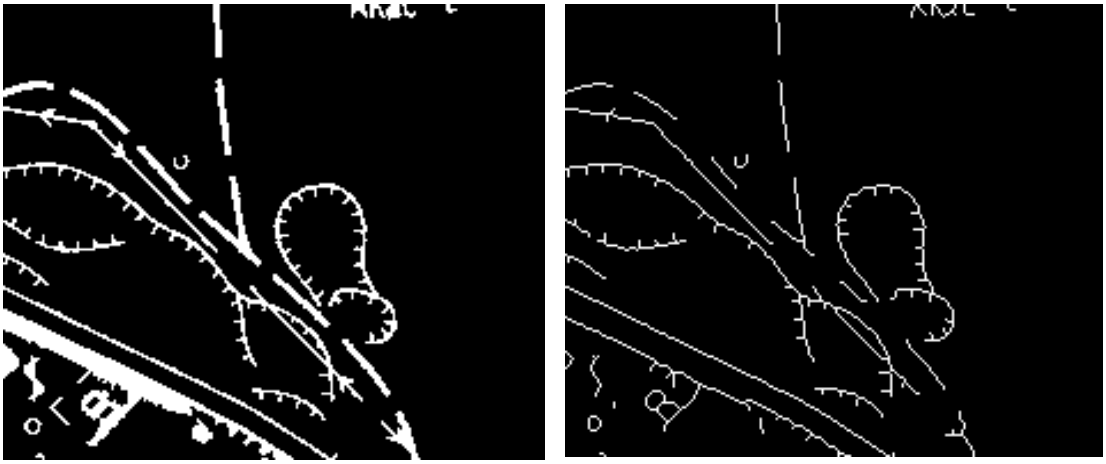
ezen lévő pixeleket töröljük a maszkról

- ha csak csúcs szomszédot találunk, akkor a pixel, amit vizsgálunk két csúcs közt van, ilyenkor ezekből készítünk törött vonalat, az ezen lévő pixeleket töröljük a maszkról

4) azok a pixelek, amik ezután maradtak és nem csúcsok, azok egy gyűrűn vannak, ezeket tetszőleges helyen elvágva törött vonalat képzünk

A töröttvolakat mindig úgy állítjuk elő, hogy a pixelközéppontokon lépkedünk mindig egyet. Így az éleink vagy 1 hosszúak lesznek vízszintesen vagy függőlegesen, vagyhosszúak átlósan. A kapott töröttvonal halmazból felépítjük a hálózati gráfot, majd minden él töröttvonalát egyszerűsítjük, figyelve arra, hogy ennek eredményeképp nehogy belemetsszünk egy másik él töröttvonalába. Az élekben azokat a töröttvonalakat, amelyeket a szintvonalak maszkjából kaptunk, megjelöljük.

A hálózati gráfban lesznek hibák, amelyek nem topológiai természetűek. Ezek abból adódnak, hogy a térképen vannak vonalszerű objektumokon és a detektált jelkulcsokon kívül más is. Ezek leginkább feliratok, amelyeket robosztus karakterfelismerő eljárásokkal lehetne kiküszöbölni. Ez a feladat önmagában is egy összetett kérdés, ami meghaladja a dolgozat témáját.



*4.7: A feketéket tartalmazó maszk. 4.8: Az ebből kapott vékonyított eredmény.*

## Területi réteg előállítása

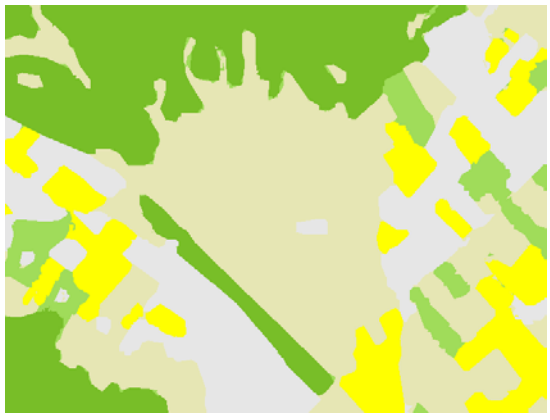
A területi réteg félépítése során az ehhez készített képet és a színeket, amelyek a területeknél előfordulhatnak, használjuk fel. Első lépésként a képet szegmentáljuk a színdetektálásra alapozott módszerrel. Az összes homogén foltot, amelyet a szegmentálás után kaptunk, egy pixellel növelünk, majd csökkentünk, így a szélek egyenetlenségeit elsimítjuk. Ezek után egy kis, pár pixel méretű területeket eltüntetjük.

Ezután a feladatunk a síkgráfhoz szükséges H töröttvonal halmaz előállítása. Ezt a

kép pixeljeinek egymásutáni feldolgozásával kapjuk. Egy képpontra a következő lépéseket végezzük el:

1) nézzük meg, hogy fel van-e dolgozva
2) ha igen, lépjünk tovább a következő pixelre
3) ha nem:
   i. a pixelből kiindulva árasszuk el a poligont egyszerű bejárással
   ii. jelöljük meg a folt összes pixeljét, hogy feldolgoztuk
   iii. a folt szélét járjuk be, a pixelek sarkait, mint rácspontokat tekintve és készítsünk belőle egy önmagába visszatérő pontsorozatot, ami a szegély lesz
   iv. keressük meg azokat a rácspontokat a szegélyen, amelyek három, vagy négy különböző színű pixel sarkán vannak
      - ha vannak ilyen pontok a szegélyen, akkor ezeknél vágjuk szét a töröttvonalat, és a kapott rész-töröttvonalakat, amelyeket korábban még nem dolgoztunk fel, tegyük bele a H halmazba
      - ha nincsen ilyen pont, akkor tetszőleges ponton vágjuk el, és úgy képezzünk belőle egy töröttvonalat és azt tegyük bele a H-ba
   v. a H halmazba az aktuális iterációban betett töröttvonalakat jelöljük meg, hogy már fel vannak dolgozva

Ez a eljárás egy jó töröttvonal halmazt ad. Mivel a szegélyt a képen rácspontok sorozataként definiáltuk, ezért a kapott töröttvonalak szigorúan két poligon határán lesznek. Kétszer azért nem fog szerepelni egyik töröttvonal sem, mert amikor egyszer feldolgozunk egyet, akkor meg is jelöljük egyúttal, így amikor a másik oldalán lévő másik poligon szegélyeként előáll, már nem tesszük be az eredménybe.

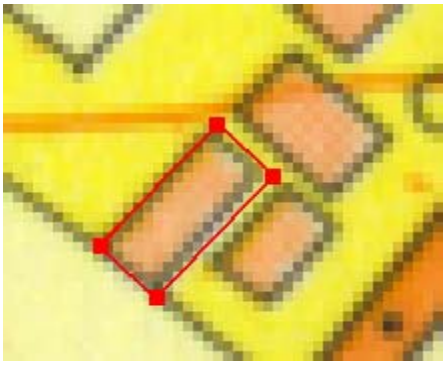


*4.9: Az előfeldolgozott raszteres kép, szegmentálás előtt. 4.10: A területeket tartalmazó állomány.*

Az eredményből ezután felépítjük a síkgráfot, duális gráfot és a befoglalási struktúrát. Az egyes területeken végigmegyünk és megnézzük, hogy milyen színnel

jelentek meg a térképen, ezt attribútumként eltároljuk. Az eredményként kapott gráf éleiben lévő töröttvonalak egyelőre csak függőlegesek és vízszintesek lehetnek, ezért azokat egyszerűsítjük távolság, majd szög alapján, figyelve arra, hogy az eredmény másik poligonba ne metsszen át.

A poligonokon végigmegyünk és kiszámítjuk a területüket, majd a legkisebb befoglaló téglalapjuk területét. Ha az eltérés arányosan kicsi és a poligon területe önmagában nem túl nagy, akkor a poligonhoz felveszünk egy leíró adatot, hogy az egy épületet jelöl.



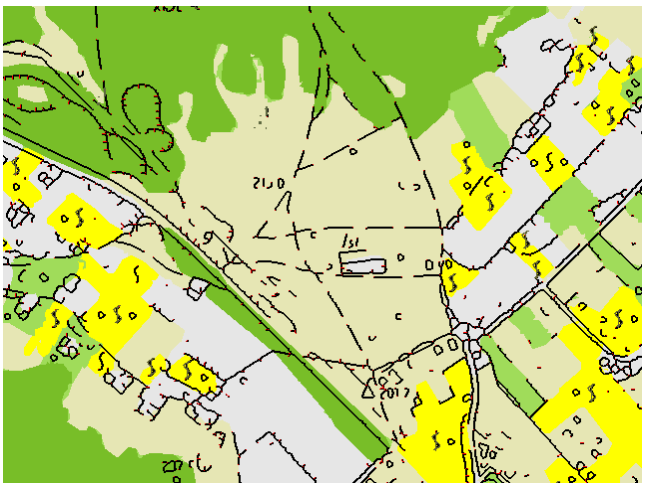
*4.11: Egy épületeket tartalmazó rész. 4.12 Az egyik épületnek megfeleltetett terület minimális lefedő téglalapja.*

## A vektorizálás eredménye

Ha ezt az eljárást végrehajtjuk, akkor eredményként a következőket kapjuk:

- Egy síkgráfot, ami a térkép területi rétegét írja le. A területekhez tároltuk azt, hogy milyen színűek, vagy ha úgy tetszik, akkor azt, hogy milyen területet jelölnek, az épületeket szintén nagy százalékban detektáltuk.
- Egy hálózati gráfot, melyben benne van az összes vonalszerű objektum és azok a dolgok, amiket a mintafelismerés során nem szűrtünk ki. A töröttvonalak egy részét megjelöltük szintvonalnak vagy szaggatott vonalnak.
- Az objektum gráf tartalmazza az összes felismert mintát pontként, attribútumban tárolva a jelentését.

Ezt az adatszerkezetet, mint az előző fejezet végén láttuk könnyen átalakíthatjuk ESRI Shape formátumra, amit majd minden térinformatikai szoftver be tud olvasni.



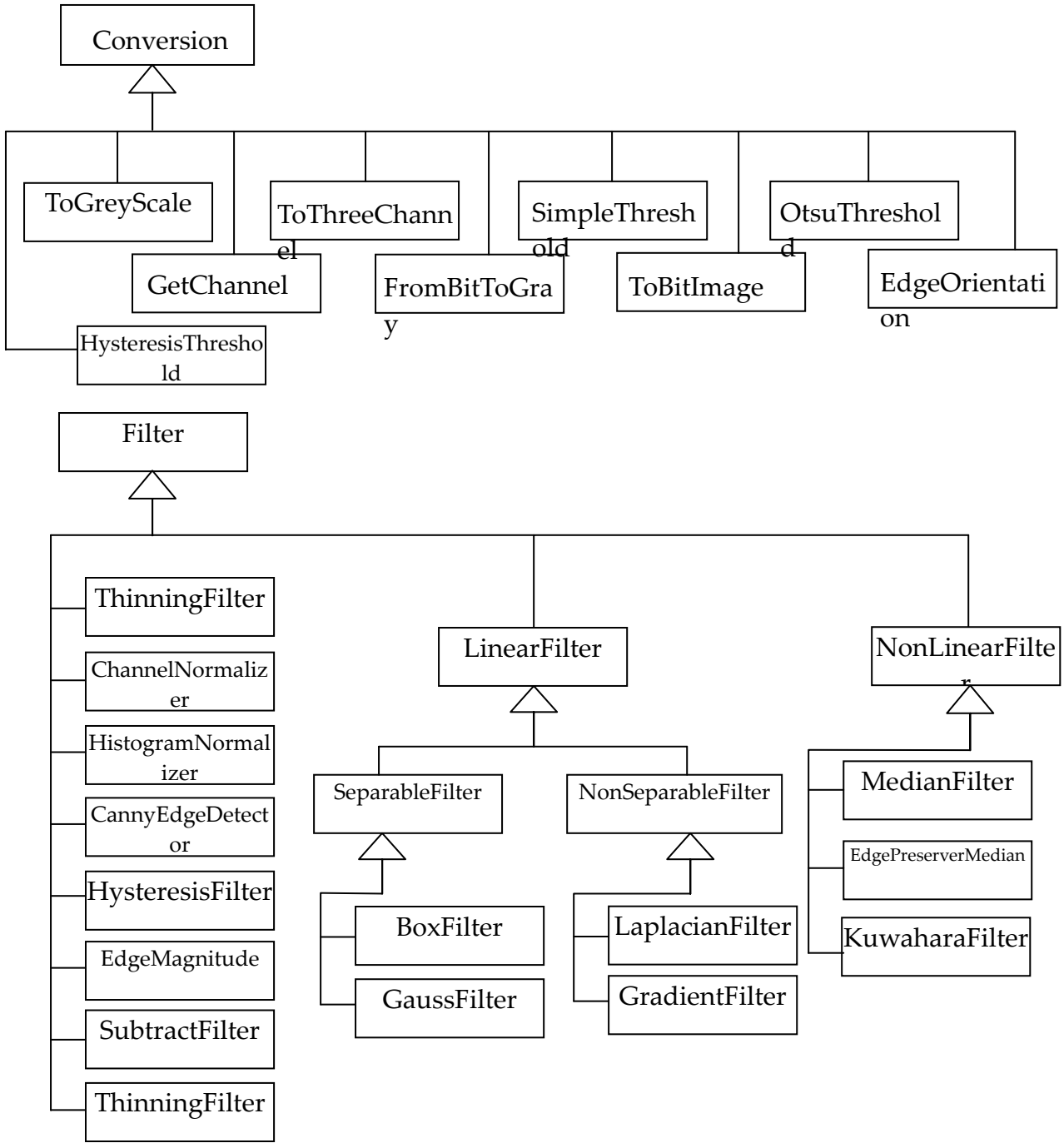
*4.13: A hálózati és területi réteg, shapefile megjelenítő programból generálva.*

# 5. fejezet: A feldolgozó program szerkezeti felépítése

## *Képmanipuláló modul*

*A képmanipuláló modul célja a digitális képelemzés legfontosabb szűrőinek, konverzióinak, összetett eljárásainak implementálása.*

### Osztályhierarchia:



## Osztályok

***Conversion<typename FromImage, typename ToImage>***
A konverziók alaposztálya csak a formát adja a többi konverzióhoz, a konvertáló eljárás üres. A konkrét konverziókat template specializációval, a FromImage és ToImage megadásával és a Convert eljárás kitöltésével lehetséges elkészíteni.

```
template <typename FromImage, typename ToImage>
class Conversion
{
public:
 Conversion() {};
 ~Conversion() {};
 virtual void Convert(const FromImage& from, ToImage& to) const {};
protected:
};
```

A Convert(FromImage, ToImage) általánosan úgy épül fel, hogy először a ToImage-t átméretezi akkorára, mint a FromImage, majd minden pixelre kiszámolja az új pixelértéket. Pl:

```
template <typename FromImage, typename ToImage>
void ToGreyScale<FromImage,ToImage>::Convert(const FromImage& from, ToImage& to) const
{
        int height = from.GetHeight(), width = from.GetWidth();
 if(height != to.GetHeight() || width != to.GetWidth())
 {
  to.Resize(height,width);
 }

 for(int i = 0; i < height; i++)
 {
  for(int j = 0; j < width; j++)
  {
   to(i,j,0) = (int)(from(i,j,0)*0.3f + from(i,j,1)*0.59f + from(i,j,2)*0.11f);
  }
 }
}
```

A konverziók, amelyek így működnek:
***ToGreyScale<Image<3,8>,Image<1,8>>***
***GetChannel<Image<3,8>,Image<1,8>>***
***ToThreeChannel<Image<1,8>,Image<3,8>>***
***FromBitToGrey<Image<1,1>,Image<1,8>>***
***SimpleThreshold<Image<1,8>,Image<1,1>>***
***ToBitImage<Image<1,8>,Image<1,1>>***

A következő konverziók implementációja eltér:
***OtsuThreshold<Image<1,8>,Image<1,1> >***
A különbség annyi, hogy egy Histogram objektumot építünk a FromImage-ből és annak a BilevelOtsuThreshold eljárásával kapjuk a küszöbértéket.

***EdgeOrientation<Image<1,8>,Image<1,64> >***
Itt egy x és y irányú GradientFilter készül, amivel az x és y irányú deriváltakat különbségi hányadossal közelítjük a imgx és imgy változóban.

```
FromImage imgx(from), imgy(from);
GradientFilter<FromImage> gradx(3.4142,'X'),grady(3.4142,'Y');
gradx.Process(imgx);
grady.Process(imgy);
```

Ezek után pixelenként egy double értéket kap a to(i,j,0) = (double)(imgy(i,j,0)/imgx(i,j,0)) képlet segítségével.

***HysteresisThreshold<Image<1,8>,Image<1,1> >***
Ebben a konverzióban egy DFS nevű privát segédfüggvény egy mélységi bejárást valósít meg. A _low és _high privát adattagok tárolják a HyteresisThreshold algoritmus alsó ill. felső határértékét. Amikor a Convert eljárásban egy a _low alá eső pixelértékhez érünk, akkor azt 0-ra állítjuk a ToImage-ben. Amikor egy _high érték fölöttit találunk, akkor az 1-es értéket kap a ToImage-ben. Amikor a két érték közti pixelre bukkanunk, akkor abból a DFS rekurzív függvénnyel megnézzük, hogy _low és _high közé eső pixelek mentén el tudunk-e jutni egy _high feletti értékhez. Ha igen, akkor 1-es értéket kap a pixel a ToImage-ben, különben 0-t. A DFS algoritmus különlegessége, hogy hatékonysági okokból a visszatérés során igaz értékek esetén az elérési útvonalat igazra állítja.

```
template <typename FromImage, typename ToImage>
bool HysteresisThreshold<FromImage,ToImage>::DFS(FromImage& img, int i, int j) const
{
 static const int dv[] = { -1, -1,  0,  1,  1,  1,  0, -1};
 static const int dh[] = {  0,  1,  1,  1,  0, -1, -1, -1};
 if(img.ValidCoord(i,j))
 {
  if(img(i,j,0) < _low){
   return false;
  }
  if(img(i,j,0) >= _high){
   return true;
  }
  else{
   for(int k = 0; k < 8; k++){
```

```
      if(DFS(img,i+dv[k],j+dh[k])){
        img(i,j,0) = 255;
        return true;
      }
      else{
        img(i,j,0) = 0;
        return false;
      }
    }
    return false;
  }
 }
 else{
  return false;
 }
}
```

***Filter<typename Image>***

A szűrések alaposztálya, hasonlóan a konverziók alaposztályához, nincs kitöltve tényleges utasétásokkal, csak az általános Filter interfészt írja le. Ez úgy alakul, hogy a Process függvénynek kell átadni a képet, amivel dolgozunk.

```
template <typename _Image>
class Filter
{
public:

 typedef _Image Image;

 Filter() {}
 virtual ~Filter() {}
 virtual void Process(Image&) const {};
 virtual void FillKernel(){};
 virtual void PrintFilter() const { std::cout << "General Image Filter" << std::endl; };
};
```

A szűrés általános alakja úgy működik, hogy 0-tól Image::spp -ig minden sávon ugyanazokat a műveleteket végezzük a kép minden pixeljére.

A Filter osztály közvetlen leszármazottai azok a szűrők, amelyek vagy csak egy pixelt vesznek figyelembe az új pixelek kiszámításakor, vagy összetett algoritmust alkalmaznak, amelyeket értelmetlen besorolni a LinearFilter és NonLinearFilter kategóriákba, mert nem kerül szóba a kernelméret.

***ChannelNormalizer<typename Image>***

A csatornanormalizálás során először a sávon kiszámoljuk a minimális és maximális intenzitásértékeket, majd a [min,max] intervallumba eső pixeleket eltoljuk a [0,max-

min] intervallumba, és felszorozzuk a $\frac{2^{Image::bps}-1}{max-min}$ értékkel.

***HistogramNormalizer<typename Image>***
A hisztogranormalizálás során minden sávra elvégezzük a csatornanormalizálást.
***EdgeMagnitude<typename Image>***
Az élkiterjedés számolásánál egy x és y irányú gradienskiszámoló szűrőt példányosítunk, amikkel kiszámoljuk egy-egy Image<1,8> típusú objektumba az x és y irányú különbségi hányadosokat, majd pixelenként kiszámoljuk az adott pixelben az adott x és y komponensű derivált hosszát.
***CannyEdgeDetector<typename Image>***
A Canny éldetektor az ismert éldetektáló algoritmust implementálja. Kiszámolja az élkiterjedést és az élorientációt, majd a nem maximális éleket kiszűri. Így egy szürkeárnyalatos képet kapunk, amiken minden él látszik.
***HysteresisThreshold<typename Image>***
Az éldetektálás utáni lépést a HysteresisThreshold konverzió példányosításával valósítja meg.
***SubtractFilter<typename Image>***
Képek kivonása, triviális implementációval: végigmegy a két képen és kivonja egymásból a pixeleket.
***ThinningFilter<typename Image = Image<1,1> >***
Az élvékonyító szűrő minden pixelre 5 feltételt vizsgál. Ha ezek igazat adnak, akkor a pixelt 0-ra állítja. Megnézi, hogy a pixel környezetében hány 1-es pixel van (CountNonZero), hányszor vált jelet a pixel környezete körbejárás során (CountTransition), valamint a szomszédos 1-es értékű pixelek elhelyezkedését. A ws tömbben vektorok vannak, amiben az előző, aktuális és következő két sor van. Így nem kell lemásolni az egész képet, tehát kevesebb segédmemóriára van szükség.

***SeparableFilter<Image>***
A szeparábilis szűrők olyan konvolúvós szűrők, melynek (2k+1)x(2k+1) méretű kernelét két vektor egyszerű szorzataként fel lehet írni. Ekkor a konvolúciót úgy is el lehet végezni, hogy egyszer a sorvektorral, egyszer az oszlopvektorral konvolváljuk a kép pixeleit.

```
template <typename Image>
class SeparableFilter :
  public LinearFilter<Image> //todo: nagyon sok minden
{
public:
  SeparableFilter(int size);
  virtual ~SeparableFilter();
  virtual void FillKernel();
  virtual void PrintFilter() const;
  int GetSize() const { return _kernelsize; }
  virtual void Process(Image& img) const;
```

```
protected:
 int _border;
 int _kernelsize;
 double _xweight,_yweight;
 double *_xarray, *_yarray;
};
```

A sorvektort az _xarray, az oszlopvektort az _yarray tartalmazza, a hozzájuk tartozó súlyértékeket az _xweight és _yweight, a vektorok méretét pedig a _kernelsize.
A SeparableFilterből két előre implementált változat a GaussianFilter és a BoxFilter. Ezen szűrőknek és más, felhasználóként esetleg hozzáadott szeparábilis szűrőknek csak az _xarray, _yarray, _kernelsize, _border változóinak kitöltését kell megadni, a Process függvényt a SeparableFilter osztálytól öröklik.

```
template <typename Image>
void SeparableFilter<Image>::Process(Image& img) const
{
 Image tmp(img);
 int xcoord,ycoord;
 int height = img.GetHeight(), width = img.GetWidth();
 int spp = img.spp;
 double* components = new double[spp];
 for(int i = 0; i < height; i++){
   for(int j = 0; j < width; j++){
     ycoord = j;
     for(int k = 0; k < _kernelsize; k++){
       xcoord = i + k - _border;
       for(int l = 0; l < spp; l++) components[l] += img.At(xcoord,ycoord,l)*_xarray[k];
      }
     for(int l = 0; l < spp; components[l++] = 0)
     tmp(i,j,l) = Nint(std::abs(components[l]/_xweight));
    }
  }

 for(int i = 0; i < height; i++)
  {
   xcoord = i;
   for(int j = 0; j < width; j++)
    {
     for(int k = 0; k < _kernelsize; k++)
      {
       ycoord = j + k - _border;
       for(int l = 0; l < spp; l++)
        components[l] += tmp.At(xcoord,ycoord,l)*_yarray[k];
      }
     for(int l = 0; l < spp; components[l++] = 0)
      img(i,j,l) = Nint(std::abs(components[l]/_xweight));
    }
  }
}
```

*NonSeparableFilter<Image>*

A nemszeparábilis szűrők azok a konvolúciós szűrők, melyeket nem lehetséges felírni két vektor szorzataként. Ekkor a teljes konvolúciós mátrixot tárolni kell. Megvalósításuk a következőképp alakul:

```
template <typename Image>
class NonSeparableFilter :
        public LinearFilter<Image>
{
public:
        NonSeparableFilter(int size);
        virtual ~NonSeparableFilter();
 virtual void FillKernel();
        virtual void PrintFilter() const;
        int GetSize() const { return _kernelsize; }
        virtual void Process(Image& img) const;
protected:
        int _border;
        int _kernelsize;
        double _weight;
        double** _kernelmatrix;
};
```

A _kernelmatrix egy kétdimenziós tömb, ami a súlyokat tartalmazza. A _weight az értéket, amivel normalizálni kell az eredményt, a _kernelsize pedig, hogy mekkora a kernel mérete.

A nemszeparábilis szűrőkből két előre beépített szűrőt készült el, a GradientFilter és a LaplacianFilter.

Ugyanúgy, mint a szeparábilis szűrőknél, itt is minden előre elkészített, vagy újonnan hozzáadott nemszeparábilis szűrőnek a privát adattagjait kell csak kitölteni, a Process minden esetben ugyanúgy néz ki: vesszük a kernelmátrix alatti pixeleket, és azoknak a megfelelő _kernelmatrix-beli súlyokkal vett átlagát számoljuk:

```
template <typename Image>
void NonSeparableFilter<Image>::Process(Image& img) const
{
 Image tmp(img);
 int sumr=0, sumg=0, sumb=0;
 int xcoord,ycoord;
 int height = img.GetHeight(), width = img.GetWidth();
 int spp = img.spp;
 double* components = new double[spp];
 for(int i = 0; i < height; i++){
  for(int j = 0; j < width; j++){
   for(int k = 0; k < _kernelsize; k++){
    for(int l = 0; l < _kernelsize; l++){
```

```
        xcoord = i + k - _border;
        ycoord = j + l - _border;
        for(int n = 0; n < spp; n++) components[n] += img.At(xcoord,ycoord,n)*_kernelmatrix[k][l];
      }
     }
     for(int n = 0; n < spp; components[n++] = 0) tmp(i,j,n) = Nint(std::abs(components[n]/_weight));
    }
   }
   img = tmp;
}
```

***NonLinearFilter<typename Image>***

A nemlineáris szűrők esetén kernelmértről van értelme beszélni, de nem egyszerű konvolúcióval számoljuk az új pixelértéket, hanem valamilyen algoritmus szerint. Ezért a nemlineáris szűrők nem örökölnek semmilyen általános algoritmust a szűrés elvégzésére.

```
template <typename Image>
class NonLinearFilter : public Filter<Image> {
public:
  NonLinearFilter(int size=0):_border(size),_kernelsize(2*size+1) {};
  virtual ~NonLinearFilter() {};
  virtual void PrintFilter() const {
    std::cout << "Kernelsize: " << _kernelsize << std::endl;
  };
  int GetSize() const { return _kernelsize; }
  virtual void Process(Image&) const {};
protected:
  int _border;
  int _kernelsize;
};
```

A _kernelsize változó tárolja a kernel méretét. Előre definiált nemlineáris szűrések:

***MedianFilter<typename Image>***

A Process eljárásban pixelértékeket egy std::vector<Image::Value> típusú vektorba tesszük, majd az std::nth_element függvénnyel kivesszük a nagyság szerint középső elemet. A szűrés műveletigénye így (img._width*img._height*_kernelsize)

***EdgePreserverMedianFilter<typename Image>***

A Process eljárásban pixelértékeket egy std::vector<Image::Value> típusú vektorba tesszük. A pixeleknek azonban csak a felét választjuk ki. Majd az std::nth_element függvénnyel kivesszük a nagyság szerint középső elemet. A szűrés műveletigénye így (img._width*img._height*_kernelsize/2)

***KuwaharaFilter<typename Image>***

A Process eljárásban létrejön négy Signature<Image::spp,Image::bps> típusú

objektum. Minden ilyen Signature a kernelablak egyik negyedéhez tartozik, az ide eső pixeleket tesszük bele. Végül megnézzük, hogy a Signature::Variance(), azaz a szórás melyik negyedben a legkisebb. Az aktuális pixelt az ebbél a pixelstatisztikából számolt átlaggal helyettesítjük.

```
template <typename Image>
void KuwaharaFilter<Image>::Process(Image &img) const
{
 Image tmp(img);
 SimpleSignature<Image::spp,Image::bps> sign[4];
 int xcoord,ycoord;
 int height = img.GetHeight(), width = img.GetWidth();
 for(int i = 0; i < height; i++)
 {
  for(int j = 0; j < width; j++){
   int resind = 0;
   for(int k = 0; k <= _border; k++){
    for(int l = 0; l <= _border; l++){
     xcoord = i + k - _border;
     ycoord = j + l - _border;
     sign[0] += img.GetPixelAt(xcoord,ycoord);
    }
   }
   … //többi negyed
   tmp.SetPixel(i,j,sign[MinInd(sign)].MeanPixel());
   for(int i = 0; i < 4; i++) sign[i].Clear();
  }
 }
 img = tmp;
}
```

## *Képkezelő modul fejlesztői dokumentációja*

A képkezelő modul fejlesztői dokumentációja azokat az információkat tartalmazza, amely a kód pontos megértését és esetleges továbbfejlesztését segítik. Ezek irányelvek, algoritmusok leírásai és megjegyzések a megvalósítási részletekkel kapcsolatban.

## Osztályok

### *Pixel<spp,bps> - Pixel.h*

A pixel osztálynak van egy kiinduló alapváltozata, ami nincs kifejtve, valamint bizonyos speciális, gyakran használt értékekre elkészített specializált template-ek.

A pixelosztály kiinduló változata nincs kifejtve, mivel a bps értéke bizonyos speciális értékeken kívül nem mondja meg egyértelműen, hogy milyen típussal kell ábrázolni az egyes sávok értékét. Tehát ha a beépített bps értékeken kívül újat akarunk hozzávenni, annak el kell készíteni a template specializációját.

```
template <int _spp, int _bps>
class Pixel {
  static const int spp = 0;
  static const int bps = 0;

  typedef Invalid Value;
  typedef Value& Reference;
  typedef const Value& ConstReference;
};
```

A template specializációk a következő speciális bps értékek esetén a Pixel::Value típus a következőképp alakul:
1 – bool
8 – unsigned char
16 – unsigned short
32 – unsigned long
64 – double
A Pixel<spp,1>, Pixel<spp,8>, Pixel<spp,16>, Pixel<spp,32> és Pixel<spp,64> ábrázolása megegyezik: egy std::vector<Value> típusú vektorban tároljuk a pixel sávonkénti értékét.
Pl:

```
template <int _spp>
class Pixel<_spp, 8> {
public:

  static const int spp = _spp;
```

```
 static const int bps = 8;

 typedef unsigned char Value;
 typedef Value& Reference;
 typedef const Value& ConstReference;

 Reference operator[](int index) {
  assert(index >= 0 && index < _spp);
  return _data[index];
 }

 ConstReference operator[](int index) const {
  assert(index >= 0 && index < _spp);
  return _data[index];
 }

private:
 Value _data[_spp];
};
```

A template specializáció ezen kívül el van készítve spp=1 -re is, mert ebben az esetben nincsen szükség vektorra a pixel ábrázolására. Mivel az Image<spp,bps> osztályban Pixel<spp,bps>::Value alaptípusú tömbben tároljuk az adatokat, ez nagy megtakarítást eredményez az egysávos képeknél és azok indexelésénél.
Pl:

```
template <>
class Pixel<1, 1> {
public:

 static const int spp = 1;
 static const int bps = 1;

 typedef bool Value;
 typedef Value& Reference;
 typedef const Value& ConstReference;

 Reference operator[](int index) {
  assert(index == 0);
  return _data;
 }
 ConstReference operator[](int index) const {
  assert(index == 0);
  return _data;
 }

 operator ConstReference() const { return _data; }
 operator Reference() { return _data; }

private:
```

```
  Value _data;
};
```

A Pixelekre az ==, !=, << és >> operátorok inline vannak elkészítve.

## *Image<spp,bps> - Image.h*

Az Image osztály ábrázolása a megfelelő Pixel osztály ábrázolására épül: az osztályon belül Pixel<spp,bps>::Value alaptípusú vektorban tároljuk sorfolytonosan a kép pixeleit. A Pixel<spp,bps>::Value típusra typedef segítségével Image::Value néven hivatkozunk. Ez a vektor védett adattagként _pixels néven szerepel az osztályban.

```
template <int _spp, int _bps>
class Image {
public:
  static const int spp = _spp;
  static const int bps = _bps;

  typedef ::Pixel<spp, bps> Pixel;
  typedef typename Pixel::Value Value;
  ...
protected:
  ...
  int _height, _width;
  std::vector<Value> _pixels;
};
```

Az indexelés során ebben a vektorban számoljuk ki a megfelelő pozíció helyét a vektorban és adjuk vissza a megfelelő értéket. Ehhez a _height és _width adattagokban tárolt értékre van szükség. A sorfolytonos ábrázolás esetén az [i,j] indexű pont elhelyezkedését a vektorban az i*_width+j képlettel lehet kiszámolni.
A pozíció és sáv érvényességét a _ValidCoord(int i, int j) és a _ValidSample(int s) függvényekkel lehet ellenőrizni.
Nem egész számokkal indexelve a képet a környező 4 pontból bilineáris interpolációval számolódik ki a CalculatePixelAt(double i, double j) által visszaadott érték.

## *Signature<spp,bps> - Pixel.h*

A pixelstatisztika osztályt egy kovarianciamátrixszal, egy összegpixellel és egy pixelszámmal ábrázoljuk.

```
template <int _spp, int _bps>
class Signature {
...
```

```
private:
 int _num;
 double _sum[spp];
 double _mul[spp][spp];
};
```

A kovarianciamátrixot a következő képlettel számoljuk: _mul[i][j] = $\sum_{k=0}^{n} p_k[i] \square p_k[j]$

Az Add függvény módosítja a pixelek számát, és kiszámolja az átlag és a kovariancamátrix változását:

```
Signature& Signature::Add(const Pixel<spp, bps>& pixel, int cnt) {
 for (int i = 0; i < spp; ++i) {
  for (int j = 0; j < spp; ++j) {
   _mul[i][j] += cnt * pixel[i] * pixel[j];
  }
  _sum[i] += cnt * pixel[i];
 }
 _num += cnt;
 return *this;
}
```

## *Histogram<Image>*

A hisztogram Image::spp darab, 2^Image::bps méretű tömbben tárolja a hisztogramokat. Így lehetőség van többsávú kép hisztogramjának tárolására is. Egy sáv hisztogramját tároló típust neve HistogramArray néven definiáltam. A hisztogramokat tároló tömb neve _harrays. A _mean tartalmazza a sávonként vett átlagot. A _signature változóban egy pixelstatisztika van.

```
template <typename Image>
class Histogram
{
public:
 ...
protected:
 typedef int HistogramArray [codomain_num];
 HistogramArray _harrays[Image::spp];
 SimpleSignature<Image::spp,Image::bps> _signature;
 int _mean[Image::spp];
 int _pixnum;
};
```

A BilevelOtsuThreshold egy küszöbölést végez Otsu algoritmusával. Ennek paraméterként meg kell adni a sávot is, amelyen dolgoznia kell.

## *MyLoadImage*

**Gdi++**

A Windowsos betöltő létrehoz egy Gdiplus::Bitmap objektumot, aminek aztán kinyeri a pixeljeinek sorfolytonosan tárolt változatát egy Gdiplus::bmData objektumba. Ebből aztán átolvassa a pixelértékeket az Image objektumba.

**Cimg**

A Cimg libraryt használó betöltő egy Cimg<unsigned char> objektumba tölti be a képet, majd ennek az indexelés operátorával lényegében egy az egyben átmásolja az Image objektumba.

## *MySaveImage*

**Gdi++ & Cimg**

A MySaveImage pontosan a MyLoadImage tükörképeként viselkedik, mindkét esetben. Ezt a függvényt minden lehetséges képtípusra külön el kell készíteni.

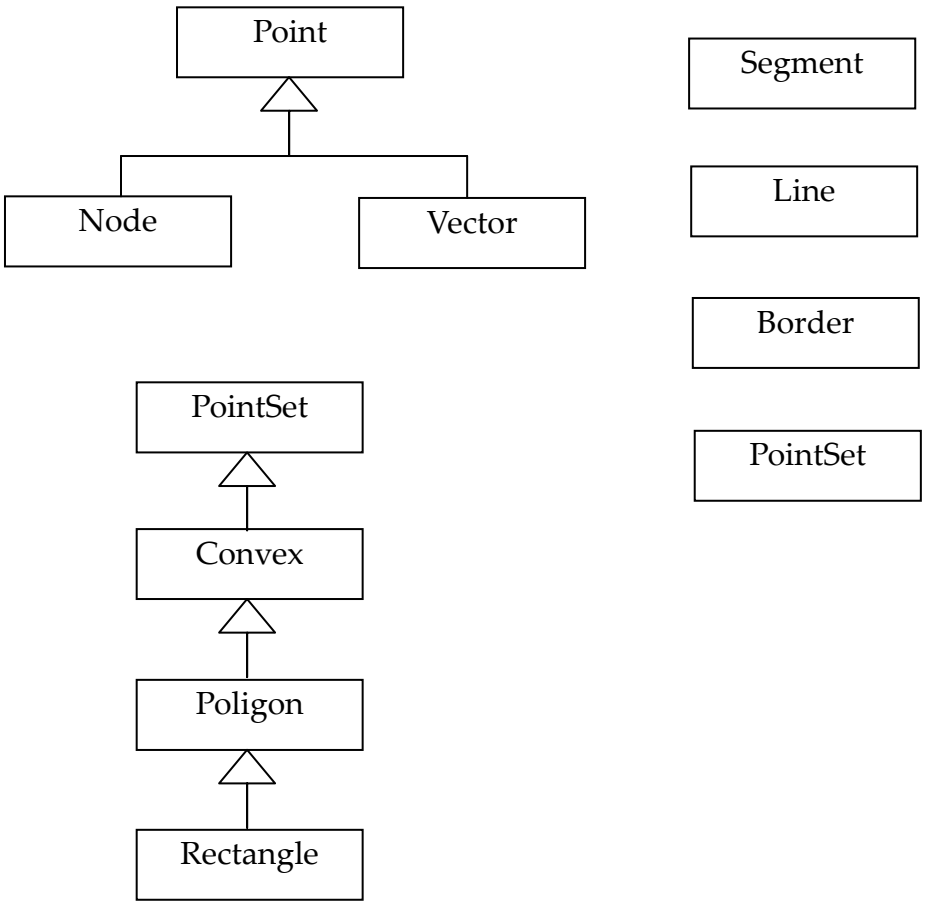
```
void MySaveImage(const Image<1,8>& image, const std::string& filename) {

 Gdiplus::Bitmap *tmpbmp = new Gdiplus::Bitmap(image.GetWidth(), image.GetHeight(), PixelFormat32bppARGB);
 Gdiplus::BitmapData bmData;
        Gdiplus::Rect *rect = new Gdiplus::Rect(0, 0, tmpbmp->GetWidth(), tmpbmp->GetHeight());
 tmpbmp->LockBits(rect, Gdiplus::ImageLockModeRead | Gdiplus::ImageLockModeWrite, PixelFormat32bppARGB, &bmData);
 int stride = bmData.Stride;
 BYTE *p = (BYTE *)((void *)bmData.Scan0);
 int nOffset = stride – image.GetWidth()*4; //atugrando sorok sorvegenkent
 std::cout << p[0] << std::endl;
 for(int x=0; x < image.GetHeight(); x++) {
  for(int y=0; y < image.GetWidth(); y++ ) {
   p[3] = 0;   // alpha
   p[2] = image(x, y, 0);
   p[1] = image(x, y, 0);
   p[0] = image(x, y, 0);
   p += 4;
  }
  p += nOffset;
 }
 tmpbmp->UnlockBits(&bmData);
```

## *Geometriai modul*

*A geometriai programmodul a geometriai entitásokat leíró osztályokat és hozzájuk kapcsolódó függvények, eljárásokat tartalmaz.*

### Osztályhierarchia




### Osztályok

***Point***

A pontot leíró osztály belső reprezentációja két double típusú adattag, _x és _y néven. Ezek mondják meg a pont x és y koordinátáit. Az operátorok a szokásos értelmezéssel, értelemszerűen vannak megvalósítva. A rendezési relációk a pontokat balról jobbra, lentről felfele rendezik.

```
class Point
{
public:
  …
protected:
  double _x,_y;
```

```
};
```

### *Node*

A Node osztály egy segédosztály, amit a Border osztály használ. A Point belső reprezentációjához csatlakozik egy bool érték, ami azt mondja meg, hogy a Node a poligonizált szegélyt közelítő poligonnak csúcsa, vagy sem.

```
class Node : public Point
{
public:
   Node():Point(),_isnode(false) {};
   ...
protected:
  bool _isnode;
};
```

### *Vector*

A helyvektorokat megvalósító osztály, megvalósítását a Point osztályból örökli. A lényegi különbség a Pointhoz képest, hogy a +,-,*, stb operátorok értelmezettek rá. Ez azért fontos, mert így hibás kifejezéseket, szemantikai tévedéseket a fordítóprogram nem enged meg. Másrészt a vektor fogalom hasznos, amikor egyenesek reprezentációjánál.

```
class Vector : public Point
{
public:
  Vector():Point() {};
  ...
  friend void operator+=(Vector& p, const Vector& q);
  friend Vector operator+(const Vector& p, const Vector& q);
  friend Point operator+(const Point& p, const Vector& q);
  friend void operator-=(Vector& p, const Vector& q);
  friend Vector operator-(const Vector& p, const Vector& q);
  friend Point operator-(const Point& p, const Vector& q);
  friend Vector operator*(const Vector& p, const double& c);
  friend void operator*=(Vector& p, const double& c);
  friend double operator*(const Vector& q, const Vector& p);
  ...
protected:
};
```

### *Line*

Az egyeneseket leíró osztály belső reprezentációja egy Point és egy Vector típusú adattagból áll. Ezek egy pontot és az egyenes irányvektorát jelölik. Hogy a szakaszokhoz hasonlóan lehessen vele dolgozni, ezért értelmezett rá a szögletes zárójel operátor, de ehelyett a GetPoint, GetVector biztonságosabban használható, mivel itt egyértelmű, hogy milyen típusú a visszatérési érték.

```
class Line
{
public:
  ...
protected:
  Point _p;
  Vector _v;
};
```

***Segment***

A szakaszokat megvalósító osztály reprezentációja egy két pontból álló tömb, amely tartalma a szakasz két végpontját jelenti. A függvények megvalósítása értelemszerű.

```
class Segment
{
public:
  ...
protected:
  Point _points[2];
};
```

***PointSet***

Pontok egy halmazát leíró osztály. A belső reprezentáció egy std::list<Point> típusú lista, amelyben a ponthalmaz pontjai tárolódnak. Erre a listára csak egy megkötés van: a _TestList(std::list<Point>&) függvény igazat adjon vissza. A PointSet osztálynál ez csak annyit ellenőriz, hogy legyen legalább 3 pont a ponthalmazban. A mechanizmust a PointSet leszármazottai öröklik. A PointSeten belül van egy minx, miny, maxx, maxy változó, amelyek az algoritmusok során kellenek. Ezek az adattagok mutable tagok, azaz megváltoztathatóak const függvényekben is.

PointSet::iterator, PointSet::const_iterator

A PointSet-en belül definiált iterátor egy std::list<Point>::iterator illetve const_iterator segítségével van megvalósítva.

A MinAreaBoundingRect() függvény a minimális lefedő téglalapot adja vissza, de ezt a Convex osztály ugyanezen metódusával számolja ki.

A PointSet legfontosabb művelete a ConvexHull() függvény, ami egy Convex típusú objektummal tér vissza, ami a ponthalmaz konvex burkát írja le.

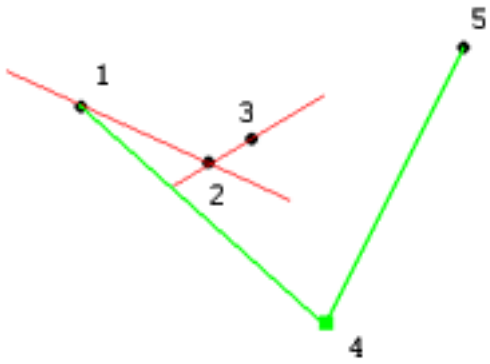
A ConvexHull függvény a következőképp működik:

- rendezi a ponthalmaz pontjait balról jobbra és fentről lefele, a Point < operátorának megfelelően
- létrehoz egy teteje és alja vermet, amiben a konvex burok két része lesz,

mindkettőbe beleteszi a rendezett pontsorozat első két elemét

- a pontokat a rendezett sorrendben feldolgozza, mégpedig úgy, hogy megnézi, hogy a következő pont az előző két pont által meghatározott egyenesnek megfelelő oldalára esik-e, és ha igen, akkor beteszi a verembe, ha nem, akkor a veremből kidobja a legfelső pontot, és megnézi, hogy az így nézett két legutolsó két pontnak megfelelő oldalán van-e és így tovább

A következő képen látszik, hogy a konvex burok alsó része 1-4-5 pontokból fog állni. Kezdetben azonban az alsó veremben 1-2-3 lesz. Ezután a 4-es a 2-3 egyenesnek nem a megfelelő oldalára esik, ezért a 3-ast kidobjuk. Az 1-2 egyenes még mindig nem jó, mert ennek sem a megfelelő oldalára esik a 4-es pont. Ezután a veremben már csak a kiinduló pont van, így az alsó burok kezdete 1-4 lesz. A felső buroknál ugyanez a helyzet.




Azt, hogy melyik oldalára esik egy pont egy egyenesnek, az IsOnLeft függvénnyel számoljuk ki. Ez lényegében az előjeles távolságból következtet az elhelyezkedésre.

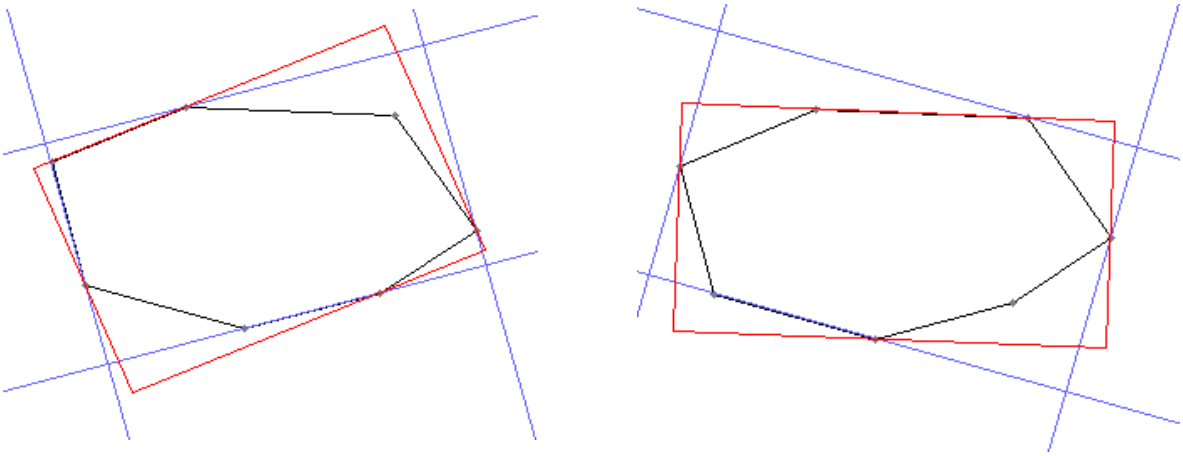
***Poligon***

A PointSet-ből származtatott osztály, a reprezentációja megegyezik. Ami a lényeges különbség, hogy nem lehet akármilyen a pontok sorrendje, valamint le lehet kérdezni a területet.

A Poligon osztályba új pontlistát tölteni a LoadList(std::list<Point>&) eljárással lehet. A LoadList a PointSetből örökölt mechanizmus szerint a _TestList(std::list<Point>&) függvénnyel leellenőrzi, hogy a kapott listában érvényes adatok vannak-e. Ez a függvény ellenőrzi, hogy 2-nél több pont legen a listában, valamint a poligon oldalai ne messék el egymást. Ha a TestList(std::list<Point>&) igazat ad, akkor a _LoadValidList(std::list<P>&) függvény betölti a listát, de előtte megnézi, hogy az előjeles terület így pozitív, vagy negatív. Ha negatív, akkor megfordítja a listát, mert ez azt jelenti, hogy a körüljárás iránya az óramutató járásával egyirányú, ilyenkor

meg kell fordítani. Ez a std::list<>::reverse() függvénnyel történik.
A területet a SignedDistance() függvény visszatérési értékével kapjuk meg. A SignedDistance-ben egy él és az x tengely közti területeket összegezzük, előjelesen, így megkapjuk a poligon területét.

***Convex***
A konvex poligonokat megvalósító osztály minden műveletét és tulajdonságát örökli a Poligontól, két változással. A LoadList(std::list<Point>&) eljárásban a _TestList(..) leellenőrzi azt is, hogy a poligon konvex-e. Ezt az IsOnLeft eljárással teszi, megnézi, hogy a pontok sorozatára igaz-e az, hogy két pont által meghatározott egyeneshez képeset a következő mindig ugyanazon az oldalon található-e. Ezt az ellenőrzést az _IsConvexList(..) függvény végzi.
A másik változás az, hogy itt van kifejtve a MinAreaBoundingRect() függvény, ami a minimális területű lefedő téglalapot adja meg egy ponthalmazra. A PointSet osztályban ez úgy történik, hogy először a ponthalmaz konvex burkát számoljuk ki, majd ennek a buroknak hívjuk meg a MinAreaBoundingRect() függvényét. A minimális területű lefedő téglalap kiszámítása a „rotating calipers" algoritmussal történik. Ennek lényege, hogy négy érintőt illesztünk a konvex burokra, balról, jobbról, lentről és fentről, rendre a legszélső ponthoz. Ezek után minden iterációban a legkisebb lehetséges szöggel elforgatjuk ezeket az érintőket, hogy egy élre simuljanak és az így kapott téglalapok területe közül vesszük a legkisebbet. Az algoritmus műveletigénye **O**(n).



Az érintőket a Convex::Caliper publikus osztály (struct) írja le. Ennek reprezentációja egy pont, amit épp érint az érintő, a következő pont, és egy listapointer, amiből a pontokat lehet kiolvasni.

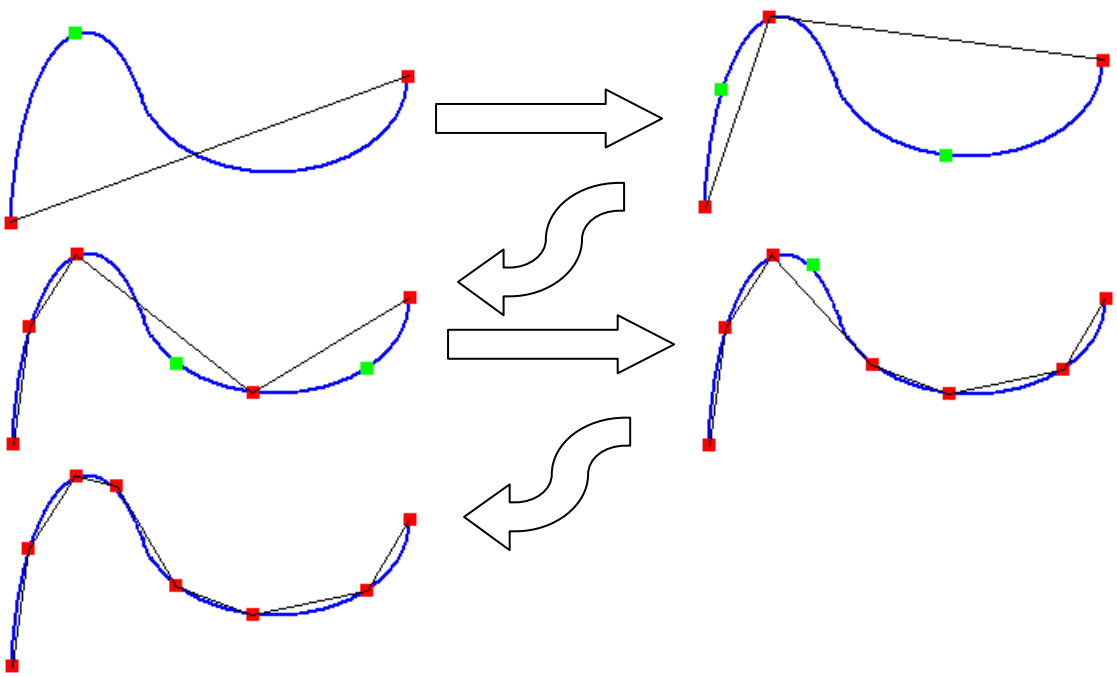
```
struct Caliper{
 Caliper() {};
 ...
 const std::list<Point>* nl;
 Line l;
 std::list<Point>::const_iterator current, next;
```

};

A Convex::ConvexHull() függvény az aktuális objektumot adja vissza eredményül, hiszen egy konvex poligonnak a konvex burka önmaga.
A _LoadSafeList védett adattag és a PointSetből elérhető, ugyanis, ott a ConvexHull() eljárásban egy biztosan megfelelő poligont állítunk elő, így azt felesleges lenne újra leellenőrizni.

***Rectangle***
Speciális poligonokat, téglalapokat megvalósító osztály. A változás annyi, hogy a _TestList azt ellenőrzi, hogy 4 pontja van-e a poligonnak illetve, hogy az oldalai merőlegesek-e egymásra. A MinAreaBoundingRect() eljárás az aktuális objektumot adja vissza.
***Blob<Image,PixelChecker>***
A Blob osztály egy példányát három adattaggal írjuk le. Egy, az Image paraméternek megfelelő típusú képpel, egy pozícióval, ami a kép egy pontját azonosítja és egy PixelChacker objektummal. A PixelChecker objektumnak kell lennie gömbölyű zárójel operátorának, ami logikai értéket ad vissza, és azt mondja meg, hogy egy pixel a blobba tartozik-e. Így a blob tulajdonképpen az a folt, amit a kiinduló pixelből a PixelCheckerrel elárasztva kapunk.
A Blob leglényegesebb művelete a GetBorderList, függvény, aminek eredményeképp egy listát kapunk a folt szélén lévő pixelekből. Ezt úgy kapjuk meg, hogy először egy bitmaszkba áttesszük azokat a pixeleket, amelyek a határon vannak, azaz benne vannak a foltban, de mellette olyan pixel van, amelyre a PixelChecker hamisat ad. Ezek után ezeket a pixeleket sorrendben bejárjuk és egy listába tesszük.

***Border***
A Border osztályban a szegély pixeljeit egy std::vector<Node> objektumban tároljuk. Az értelemszerűen működő műveleteken túl a legfontosabb művelet a Poligonize(), mert ezzel térünk át raszteres jellegű formátumról teljesen geometriaira.
A poligonizálást rekurzív algoritmussal valósítjuk meg: a sorrendben első és középső, valamint középső utáni és utolsó pixelek által meghatározott görbe vonalra meghívjuk a _Poliline függvényt. A Poliline függvény végigmegy a két határ közti pixeleken, kiválasztja a legtávolabbit. Ha ez egy határnál messzebb van a két pixel által meghatározott egyenestől, akkor félbevágja a vonalat, középen elhelyez egy csúcsot, úgy, hogy a Node _isnode tagját igazra állítja, majd a két részre meghívja a _Poliline függvényt:



***PointDrawer<Image>***
Kirajzol egy pontot, összesen 9 pixelre: a ponthoz legközelebbi pixelre és annak 8 szomszédjára. Értelemszerű implementáció.

***PointPureDrawer<Image>***
Kirajzol egy pontot: a ponthoz legközelebbi pixelt állítja a kívánt színre. Értelemszerű implementáció.

***P2PSimpleDrawer<Image>***
Kirajzolja egy szakasz két végpontját összekötő egyenest. A két végpontot nem rajzolja ki. Az iránytangensből következtet az egyenes állására, és aszerint rajzolja ki az egyenest.

***P2PDottedDrawer<Image>***
Kirajzolja egy szakasz két végpontját összekötő egyenest, szaggatott vonallal. A két végpontot nem rajzolja ki. Ugyanúgy működik, mint a SimpleDrawer, de a dotlength változóban tárolja, hogy éppen kell-e kirajzolni pontot vagy sem. Ha a dotlength nagyobb, mint 0, akkor rajzol, ha kisebb, akkor nem. Ha a dotlength eléri a -2-t, akkor visszaáll 3-ra.

***SegmentDrawer<Image, PDrawer, P2PDrawer>***
A template paraméterben kapott PDrawer típusú kirajzolóval kirajzolja egy szakasz két végpontját, az összekötő egyenest pedig a kapott P2PDrawer típusú kirajzolóval.

Értelemszerű implementáció.

***PoligonDrawer<Image, PDrawer, P2PDrawer>***
A template paraméterben kapott PDrawer típusú kirajzolóval kirajzolja egy poligon pontjait, az éleket pedig a kapott P2PDrawer típusú kirajzolóval. Értelemszerű implementáció, a Poligon::const_iterator osztályt használja.

***PointSetDrawer<Image, PDrawer>***
A template paraméterben kapott PDrawer típusú kirajzolóval kirajzolja egy ponthalmaz pontjait. Értelemszerű implementáció, a PointSet::const_iterator osztályt használja.

## Függvények

***double Angle(const Vector p, const Vector q)***
A p és q vektorok által közbezárt szöget adja vissza radiánban. Az std::atan2 függvényt használja a kiszámoláshoz. Kiszámolj a q vektor és az x tengely által bezárt szöget és ebből kivonja a p vektor és x tengely által közbezárt szöget.

***Point Cut(const Line& e, const Line& f)***
Egy Point típusú objektumban visszaadja az e és f egyenesek metszéspontját (ha létezik). Ehhez megoldja a két egyenes egyenletéből adódó egyenletrendszert, így megkapja azt a pontot, mely mindkét egyenletet megoldja.

***Point Cut(const Segment& s, const Segment& t)***
Egy Point típusú objektumban visszaadja az e és f egyenesek metszéspontját (ha létezik). Ehhez megoldja a két szakasz egyenesei által meghatározott egyenletrendszert, majd leellenőrizni, hogy ez a pont rajtavan-e mindkét szakaszon.

***double SignedDistance(const Line l, const Point p)***
Az l egyenes és p pont előjeles távolságát adja meg. Az előjel mondja meg, hogy a pont az l jobb, vagy baloldalán fekszik. Ehhez a függvénytáblázatban található képletet használja.

***double Distance(const Line l, const Point p)***
Az l egyenes és p pont távolságát mondja meg. Ehhez a SignedDistance függvényt használja, ennek az abszolútértékét adja vissza.

***double Distance(const Point p, const Point q)***
A p és q pontok távolságát adja meg, a szokásos számolási móddal.

***bool ExistsCut(const Segment& s, const Segment& t)***

Megmondja, hogy az s és t szakaszok metszik-e egymást. Ehhez a metszéspont koordinátáit ellenőrzi le.

***Line Median(const Line e, const Line f)***
Az e és f egyenesek szögfelező egyenesét adja vissza egy Line típusú objektumban. Ehhez kiszámolja a két egyenes által közbezárt szöget és annak felével elforgatja az egyik egyenest annak Rotate(double) tagfüggvényével.

***bool Parallel(const Line& e, const Line& f)***
Egy logikai értékben visszaadja, hogy az e és f egyenesek párhuzamosak-e. Ezt úgy számolja ki, hogy veszi az egyik egyenes irányvektorának és a másik normálvektorának skaláris szorzatát. Ha ez nullával egyenlő, az azt jelenti, hogy a két egyenes párhuzamos.

***Point Projection(const Line& e, const Point& p)***
Egy Point típusú objektumban visszaadja a p pont e egyenesre eső merőleges vetületét. Ehhez a p pont és az egyenes normálvektora által meghatározott egyenessel veszi az eredeti egyenes metszéspontját.

# 6. fejezet: A program fejlesztői felülete

## Képmanipuláló modul

*A képmanipuláló modul célja a digitális képelemzés legfontosabb szűrőinek, konverzióinak, összetett eljárásainak implementálása.*

### *Konverziók képtípusok között*

Megvalósító alaposztály: Conversion<typename FromImage, typename ToImage>
Az osztály template paraméterei mondják meg azt, hogy egy adott konverzió milyen képtípusból milyet állít elő. A konverziók mind a Conversion<FromImage,ToImage> osztály leszármazottai. Minden egyes konkrét konverzió template paraméterei specializáltak, hiszen az algoritmusoknak csak adott típusból adott típusba való képzés esetén van értelmük.
A használat általános módja:

*template <typename FromImage = Image<3,8>, typename ToImage = Image<1,8> >*
*...*
*SpecializedConversion<> conv;*
*Megvalósított konverziók listája*[15]*:*
*Image<3,8> colorimg;*
*Image<1,8> grayimg;*
*…*
*conv.Convert(colorimg,grayimg);*

### Színesből szürkeárnyalatossá alakítás

Megvalósító osztály: ToGreyScale<Image<3,8>,Image<1,8> >
A szürkeárnyalatos képet a színes kép egyes pixeljeinek piros, zöld és kék komponensének konvex kombinációjaként számítja ki, rendre 0.3, 0.59, 0.11 súlyokkal.

15 *A használat módját csak ott részletezem, ahol az eltér az általános formától.*

## Színes kép egy csatonájának leválasztása

Megvalósító osztály: GetChannel<Image<3,8>,Image<1,8> >
A szürkeárnyalatos képet a konstruktorban megadott sáv intenzitásaival tölti ki, tehát például egy színes kép zöld sávjának értékeit a következő módon kaphatjuk meg:

```
GetChannel<> get(1); //zöld csatorna kiválasztása
Image<3,8> colorimg;
Image<1,8> grayimg;
…
get.Convert(colorimg,grayimg);
```

## Szürkeárnyalatos kép „színessé” alakítása

Megvalósító osztály: ToThreeChannel<Image<1,8>,Image<3,8> >
Egy szürkeárnyalatos képből úgy készít „színes” képet, hogy minden csatornába a szürkeárnyalatos képet másolja. Valójában a képen új információ nem keletkezik, csak új formában tároljuk ugyanazt az adatot.

## Bitkép „szürkeárnyalatossá” alakítása

Megvalósító osztály: FromBitToGrey<Image<1,1>,Image<1,8> >
Egy bitképből úgy készít „szürkeárnyalatos” képet, hogy az új képen minden pixelnek, ami az eredeti képen *true* értéket vett fel, ott 255-öt, ahol pedig *false* értéket, ott 0-t ad értékül. Valójában a képen új információ nem keletkezik, csak új formában tároljuk ugyanazt az adatot.

## Szürkeárnyalatos kép küszöbölése:

Ezek a konverziók mind szürkeárnyalatos képekből készítenek bitképeket, úgy, hogy egy alkalmas küszöbértéknél sötétebb pixeleket *false*, világosabb pixeleket pedig *true* értékre állít be.
Megvalósító osztályok:
***SimpleThreshold<Image<1,8>,Image<1,1> >***
A kívánt küszöbértéket a konstruktorban állíthatjuk be

```
SimpleThreshold<> simple(155); //a küszöb most 155
Image<1,1> bitimg;
Image<1,8> grayimg;
…
simple.Convert(grayimg,bitimg);
```

***ToBitImage<Image<1,8>,Image<1,1> >***
A simplethreshold egy változata, ahol a küszöb mindig *128*. Logikailag ez az az osztály, amely a FromBitToGrey konverzió inverze.

***OtsuThreshold<Image<1,8>,Image<1,1> >***
Az Otsu-féle küszöbölési eljárást megvalósító konverzió.[16]

## Éldetektáláshoz élek normálvektorának meghatározása.

Megvalósító osztály: ***EdgeOrientation<Image<1,8>,Image<1,64> >***
A Canny éldetektáló algoritmusban szükséges normálvektorok kiszámítására alkalmas szűrő. Az eredménye egy speciális, egysávos kép, amelyben minden pixel egy *double* értéket tartalmaz.

### *Szűrési eljárások*

Megvalósító alaposztály: ***Filter<typename Image>***
Alapvetően minden olyan algoritmust, ami egy kép alapján kiszámol egy új, ugyanolyan típusú képet, a Filter bázisosztályból származik. Az összes ilyen szűrő hatását a Filterből örökölt Process virtuális függvény felüldefiniálásával valósítottam meg. Így, minden szűrési eljárást a következő módon kell használni:

```
Image<3,8> img;
MyFilter<Image<3,8> > f;
f.Process(img);
```

## Csatornanormalizálás

Megvalósító osztály: ***ChannelNormalizer<typename Image>***
A csatornanormalizáló szűrő feladata, hogy egy színcsatornát kontrasztosabbá tegyen, a hisztogramot a *0*-ba eltolva és széthúzva *255*-ig. A konstruktorban vagy a SetChannel eljárásban be kell állítani, hogy melyik csatornát akarjuk normalizálni. Minden képtípuson működik, de igazából Image<3,8> típusú képeknél van értelme.

## Hisztogram normalizálás

Megvalósító osztály: ***HistogramNormalizer<typename Image>***
A hisztogram normalizálás egy kép kontrasztosítására szolgál: az összes csatornán elvégzi a csatornanormalizálást.

16 *N.Otsu: A threshold selection method from gray level histograms*

## Élkiterjedés kiszámolás

Megvalósító osztály: ***EdgeMagnitude3Ch<typename Image>***
A Canny éldetektálás egyik lépését, az élkiterjedés kiszámolását végzi el.

## Canny éldetektáló szűrő

Megvalósító osztály: ***CannyEdgeDetector<typename Image>***
A Canny éldetektáló algoritmust megvalósító szűrő. Eredménye a minimális élek elnyomása után előállt eredmény, tehát a „hysteresis threshold" lépést nem végzi el

## Hysteresis threshold

Megvalósító osztály: ***HysteresisThreshold<typename Image>***
A Canny éldetektor eredményeként előálló eredményt küszöböli a konstruktorban megadott két érték alapján.

## Képkivonás

Megvalósító osztály: ***SubtractFilter<typename Image>***
A képet a konstruktorban megadó eltolással kivonja önmagából.

### *Lineáris szűrők*

A lineáris, vagy konvolúciós szűrők közé azokat a szűrőket soroljuk, amelyek egy *(2k+1)x(2k+1)* méretű kernelmátrixban található súlyok alapján veszik a kernel alatt található pixelek súlyozott átlagát. A lineáris szűrőket két osztályba soroljuk: a szeparábilis és nem szeparábilis szűrők osztályába. A szeparábilis szűrők olyan lineáris szűrők, melyeknek kernelmátrixa előáll két, *2k+1* méretű vektor szorzataként. A nem szeparábilis szűrők azok a lineáris szűrők, melyekre ez a feltétel nem teljesül. Ezt az osztályozást hatékonysági okokból érdemes elvégezni.

### *Szeparábilis szűrők*

## Egyszerű átlagoló szűrő

Megvalósító osztály: ***BoxFilter<typename Image>***
A BoxFilter a legegyszerűbb átlagoló szűrő, az új pixelt a kernel aktuális pozíciója alatti pixelek átlagaként számolja ki.

## Gauss életlenítés

Megvalósító osztály: ***GaussFilter<typename Image>***

Simító szűrő, a Gauss haranggörbét közelítő értékekkel a kernelben. 2 hatványokkal közelíthető, így nagyon gyorsan számolható. A konstruktorában megadhatóak a kívánt Gauss haranggörbe paraméterei.

### *Nem szeparábilis szűrők*

## Laplace operátor

Megvalósító osztály: ***LaplacianFilter<typename Image>***
A Laplace operátort megvalósító szűrő. A konstruktorban választhatunk a leggyakoribb kernelek közül. Fix kernelméretű.

## Első deriváltakat közelítő szűrő

Megvalósító osztály: ***GradientFilter<typename Image>***
A kép, mint felület x és y irányú deriváltjait a különbségi hányadossal közelíti. A kernel a következő módon alakul:

$G_x$

$\frac{1}{p}$

| | | |
|---|---|---|
| $-1$ | $0$ | $1$ |
| $2-p$ | $0$ | $p-2$ |
| $-1$ | $0$ | $1$ |

$G_y$

$\frac{1}{p}$

| | | |
|---|---|---|
| $-1$ | $2-p$ | $-1$ |
| $0$ | $0$ | $0$ |
| $1$ | $p-2$ | $1$ |

A szűrő konstruktorában meg kell adni a *p* paramétert és azt, hogy melyik változó szerinti deriváltat közelítünk.

```
Image<3,8> imgx, imgy;
...
GradientFilter<Image<3,8> > g_x(2,'X'),g_y(2+std::sqrt(2),'Y');
g_x.Process(img);
g_y.Process(img);
```

### *Nem lineáris szűrők*

## Medián szűrő

Megvalósító osztályok:
***MedianFilter<typename Image>***
***EdgePreserverMedianFilter<typename Image>***
Statisztikai simító szűrő, a kernel alatt található pixeleket intenzitás szerinti mediánját veszi új intenzitásértékként. Az élmegőrző változat annyival tud többet

ennél, hogy minden csak a pixelek felét nézi végig, és mindegyik pixelt összehasonlítja a középpontra szimmetrikusan elhelyezkedő párjával. A pontpárból azt választja, amelyiknek az intenzitása a középponthoz közelebb van. Mindkettő konstruktorában meg kell adni a kernel méretét.

## Kuwahara-féle szűrő:

Megvalósító osztály: ***KuwaharaFilter<typename Image>***
Statisztikai simító szűrő, mely a kernelt négy szimmetrikus részre bontja. Minden ilyen részben kiszámolja a pixelek tapasztalati szórásnégyzetét, és azt a negyedet választja, ahol ez a legkisebb. Konstruktorában meg kell adni a kernel méretét.

## Élvékonyító szűrő

Megvalósító osztály: ***ThinningFilter<typename Image = Image<1,1> >***
A skeletonizáció a fekete-fehér képeken az objektumok középtengelyének meghatározása. A középtengely az objektum kontúrjától egyforma távolságban lévő pontok. Az élvékonyító szűrő eredményeként előáll egy bitképen lévő fehér színű objektumok középvonala.

# Képkezelő modul

*A képkezelő modul célja egyrészt egy általános képosztály biztosítása, amely támogatja a különböző képtípusokat, azok pixeljeit, másrészt olyan függvények biztosítása, melyekkel képeket lehet beolvasni.*

## Pixelosztály

Megvalósító osztály: ***Pixel<spp,bps>***
Egy pixelt megvalósító osztály, mely 2 egész template paraméterrel rendelkezik. Az spp jelenti a „sample per pixel”-t, a bps pedig a „bits per pixel”-t. Az első paraméter azt jelenti, hogy hány sávos a pixel, a második pedig azt, hogy az egyes sávokat hány biten ábrázoljuk Így egy Pixel<3,8> paraméterekkel példányosított kép egy 3 sávos, 256 színű (színes) pixelt jelent. Egy pixelt lehet a sávjai szerint indexelni.

```
Pixel<3,8> colorpix; //színes pixel
Pixel<1,8> graypix; //szürkeárnyalatos pixel
Pixel<1,1> bit; //bitpixel
colorpix[0] = 255; //első sáv beállítása
```

## Képosztály

Megvalósító osztály: ***Image<spp,bps>***
Ez az osztály 2 egész template paraméterrel rendelkezik. Az spp jelenti a „sample per pixel”-t, a bps pedig a „bits per pixel”-t. Az első paraméter azt jelenti, hogy hány sávos a kép, a második pedig azt, hogy az egyes sávokat hány biten ábrázoljuk Így egy Image<3,8> paraméterekkel példányosított kép egy 3 sávos, 256 színű képet (színes képet jelent).

```
Image<3,8> colorimg; //színes kép
Image<1,8> grayscaleimg; //szürkeárnyalatos kép
Image<1,1> bitimg; //fekete-fehér kép
```

Szolgáltatások:
Konstruktorok, paraméterek lekérdezése, átméretezés, többféle indexelési lehetőség, képrészlet kivágása.

Konstruktorok: szélesség és magasság megadásával, illetve egy már létező kép megadásával (copy konstruktor)

```
Image<3,8> img(10,20); //konstruktor
```

```
Image<3,8> imgcopy(img); //copy konstruktor
```

Paraméterek lekérdezése: szélesség, magasság lekérdezése.

```
int w = img.GetWidth();
int  h = img.GetHeight();
```

Indexelési lehetőségek: A következő indexelési módszereket hatékonysági szempontból csökkenő, hibatűrés szempontjából növekvő sorrendben sorolom fel:

- ellenőrzés nélküli indexelés gömbölyű zárójellel vagy a GetPixel függvénnyel
- ellenőrzéses indexelés, akár a kép határain kívülre az At vagy a GetPixelAt függvénnyel, ez a képen kívül a legközelebbi képpontot adja meg
- bilineáris interpolációval kiszámolt pixelérték a CalculatePixelAt függvénnyel

```
img(3,3,0) = 128; //érték elérése gömbölyű zárójel operátorral, a sáv megadásával
Pixel<3,8> p = img.GetPixelAt(-1,-1); //egész pixel elérése a GetPixelAt függvénnyel
Pixel<3,8> q = img.CalculatePixelAt(3.2,3.14); //bilineáris interpoláció
```

Képrészlet kivágása a Cut függvénnyel, megadva a kezdőpontot és a határokat.

```
Image<3,8> img(50,50),imgcut;
img(1,1,0) = 255;
imgcut = img.Cut(0,0,10,10); //imgcut -ba az img (0,0) pixelétől vett 10x10es része
```

## Pixelstatisztika

Megvalósító osztály: ***Signature<spp,bps>***
Az osztály template paramétere azt mondja meg, hogy milyen típusú pixelekhez tartozik a statisztika.

Szolgáltatások:
A statisztikához hozzá lehet adni és el lehet venni pixeleket és le lehet kérdezni az aktuális pixelhalmaz tapasztalati várható értékét, szórását és a sávok közti kovarianciát.

Létrehozás: paraméter nélküli konstruktorral.
Pixel hozzáadása és elvétele: az Add függvénnyel vagy a + operátorral lehet hozzáadni egy pixelt, a – operátorral pedig elvenni. Az Erase függvénnyel n-szer lehet kivonni egy pixelt a statisztikából.
Statisztikák lekérdezése: A Mean, Variance, Covariance függvényekkel.

```
Image<3,8> img;
...
Signature<3,8> sig;
sig += img.GetPixel(10,10);
double cov = sig.Covariance(0,2), mean = sig.Mean(1);
Pixel<3,8> p = sig.Mean();
```

## Hisztogram

Megvalósító osztály: ***Histogram<typename Image>***
Az osztály template paramétere mondja meg, hogy milyen típusú képhez tartozik a hisztogram.
Szolgáltatások:
A hisztogram felépíthető egy konstruktorban kapott képből, illetve függvényhívással is, lehet benne sáv és intenzitás szerint indexelni, így megkapni egy intenzitás előfordulásainak számát illetve statisztikák lekérdezésére és sávonkénti küszöbölésre van lehetőség.

Hisztogram felépítése: konstruktor paramétereként megadott képpel, vagy a BuildHistogram eljárással.
Indexelés: gömbölyű zárójellel, az első paraméterként a sávval, másodikként pedig az intenzitással.
Statisztikák: átlag lekérdezése egy sávban a Mean függvénnyel lehetséges, a szórás az i. sávban pedig a Variance függvénnyel.
Küszöbölés: a BilevelOtsuThreshold eljárás az i-edik sávban számolt Otsu-féle küszöbölési algoritmus értékét adja.

```
int num;
double variance,mean;
Image<3,8> cimg;
Image<3,8> gimg;
Histogram<Image<3,8> > h(cimg);
num = h(0,192);
variance = h.Variance();
mean = h.Mean(0);
Histogram<Image<1,8> > h1;
h1.BuildHistogram(gimg);
int thresholdval = h1.BilevelOtsuThreshold(0);
```

## Képbetöltő és képelmentő eljárások

Megvalósító függvények: ***MyLoadImage, MySaveImage***

A MyLoadImage függvény paramétere egy fájlnév és az az Image<spp,bps> objektum, amelybe a fájl tartalmát bele szeretnénk tölteni.
A MySaveImage függvény paramétere egy elmenteni kívánt képobjektum és a fájlnév, amellyel menteni szeretnénk.
A jelenlegi megvalósítás a betöltésnél színes képeket támogat, Linux és Windows alatt. Windows alatt szükség van telepített gdi+ dll-re, Linux alatt pedig a Cimg függvénykönyvtárra. Egyéb platformokon ezek a függvények nem elérhetőek, de az egész programcsomagból csak ezeket az eljárásokat kell megírni, ha valaki szeretné új környezetben használni.
A használat módja:

```
Image<3,8> img;
MyLoadImage(img,"kep.bmp");
img(img.GetWidth()/2,img.GetHeight()/2,0) = 255;
MySaveImage(img,"ujkep.bmp");
```

# Geometriai modul

*A geometriai modul célja, hogy geometriai entitásokat modellező objektumokat, eljárásokat és műveleteket biztosítson, valamint raszteres adatok geometriai interpretációjára való áttéréshez biztosítson keretet. A geometriai modul összes objektuma és eljárása a Geometry névtérben van, és a Geometry.h forrásfájl használatával érhető el. Ehhez kapcsolódóan a modul támogatja a képszintézist is, azaz az objektumok megjelenítését a képeken.*

# Geometriai osztályok

A geometriai osztályok arra szolgálnak, hogy a raszteres adatok geometriai interpretációja során szükséges entitásokat leírják.

## Pont

Megvalósító osztály: ***Point***
A legegyszerűbb geometriai entitást, a pontot megvalósító osztály.
Szolgáltatások:
Konstruktorok: a default konstruktor (0,0) koordinátákkal hozza létre a pontot. Ezen kívül lehetséges két paraméterben, vagy az Image.h-ban definiált Position objektum segítségével megadni az x és y koordinátákat. Copy konstruktor és egyenlőség operátor is értelmezett a Point-ra. A pontot beolvashatjuk standard inputról is.
Operátorok: A szögletes és gömbölyű operátorral lehetséges indexelni egy Point objektumot, így érhetjük el egy pont x és y koordinátáit. Értelmezettek a következő operátorok: <<, >>, ==, !=. Ezen kívül értelmezettek a <, <=, >, >= operátorok is, elsődlegesen az x koordinátán, másodlagosan az y koordinátán értelmezve a rendezést.
Műveletek: a pontot elforgathatjuk az origó körül, a Rotate(angle) eljárás és a GetRotate(angle) függvény segítségével. Az első az adott pontot forgatja el, a második pedig egy új pontot ad vissza, az elforgatott értékekkel. Speciálisan 90 fokkal a Rotate90() eljárás és GetRotate90() függvény ugyanezt teszi. A ToGrid() eljárás a legközelebbi rácspontra mozgatja a pontot. A Distance(Point) függvény segítségével a pont távolságát mondhatjuk meg egy másik ponttól.
A használat módja:

```
Geometry::Point p, q(3.14,2), r(q);
p = q;
p.Rotate90();
p.ToGrid();
double dist = p.Distance(q);
std::cout << p;
```

## Vektor

Megvalósító osztály: ***Vector***
A helyvektorokat megvalósító osztály. Örökli az összes szolgáltatást a Point osztályból, de ezen kívül biztosítja a helyvektorokra értelmezett műveleteket is.
Szolgáltatások:
Konstruktorok: A default konstruktor egy nullvektort hoz létre. Ezek kívül lehetőség van a helyvektor koordinátáinak megadására vagy két szám vagy egy Point típusú objektum segítségével. A vektort megadhatjuk két pont segítségével, ekkor az első pontból másodikba mutató vektort kapjuk. Ezen kívül van copy konstruktor és egyenlőség operátor is. A vektorokat beolvashatjuk standard inputról is.
Operátorok: a Point-nál megismerteken kívül értelmezettek a következő operátorok is: +, +=, -, -=, *, *=, a szokásos értelemben.
A használat módja:

```
Geometry::Point p(1,2), q(2,3);
Geometry::Vector v(p), w(p,q), u = v+w;
double mul = p.Distance(q);
std::cout << v;
```

## Egyenes

Megvalósító osztály: ***Line***
Az egyeneseket egy pontjuk és irányvektoruk segítségével megvalósító osztály.
Szolgáltatások:
Konstruktorok: a default konstruktor egy origóból induló, (1,1) helyvektorú egyenest hoz létre. Ezen kívül lehetőség van az egyenes megadására két pontjával, valamint egy pontjával és irányvektorával. Értelmezett továbbá az értékadás operátor és a copy konstruktor is. Az egyenest beolvashatjuk standard inputról is.
Operátorok, műveletek: Az egyenes indexelhető, szögletes és gömbölyű zárójellel. Előbbivel a pontját, és a pontjának a helyvektorral való eltoltját kaphatjuk meg, míg utóbbi két paramétert kapva indexeli tovább a megfelelő pontokat. Ez azért hasznos, mert gyakran van szükség egy egyenes két pontjára, illetve azok koordinátáira. Az egyenest el is lehet forgatni, a Rotate(angle) függvény segítségével, valamint meg lehet kérdezni, hogy egy pont az egyenesen van-e.
A használat módja:

```
Geometry::Point p(1,2),q(3,4);
Geometry::Vector v(0,1);
Geometry::Line e(p,v),f(p,q);
double mul = p.Distance(q);
if(f.IsOnLine(q)) std::cout << "igen" << std::endl;
std::cout << e;
```

## Szakasz

Megvalósító osztály: ***Segment***
A szakaszokat megvalósító osztály.
Szolgáltatások:
Konstruktorok: a default konstruktor egy (0,0) és (0,1) közt vezető szakaszt hoz létre. Így nem sérül az osztály invariánsa, hogy a szakasz két végpontja nem egyezhet meg. Ezen kívül a szakasz megadása lehetséges a két végpontjával. Értelmezett a copy konstruktor és az értékadás operátor is. A szakaszt beolvashatjuk standard inputról is, a két pont megadásával.
Műveletek: Egy szakaszból létre lehet hozni egy egyenest a Segment2Line függvény segítségével. Meg lehet kérdezni az IsOnSegmentLine() függvénnyel egy pontról, hogy a szakasz egyenesére esik-e valamint az IsInSegment() függvénnyel azt, hogy a szakaszon rajta van-e. A Length() függvény a szakasz hosszát mondja meg.
Használat módja:

```
Geometry::Point p(1,2),q(3,4),r(2,3);
Geometry::Segment s(p,q);
double len = s.Length();
if(s.IsInSegment(r)) std::cout << "igen" << std::endl;
std::cout << s;
```

## Ponthalmaz

Megvalósító osztály: ***PointSet***
Pontok egy halmazát leíró osztály. A pontokat listában tárolja, de nem tesz semmilyen megkötést arra nézve, hogy milyen sorrendben tárolja a pontokat. Ez a legáltalánosabb osztály, aminek értelme van kiszámolni a konvex burkát.
Szolgáltatások:
Konstruktorok: a default konstruktor egy üres ponthalmazt hoz létre. A ponthalmaz egy std::list<Geometry::Point> típusú listával is fel lehet tölteni akár a konstruktorból, akár a LoadList művelet segítségével.
Műveletek: az Add(Point p) művelettel újabb pontot lehet hozzáadni a ponthalmazhoz. A ConvexHull() függvény segítségével egy konvex poligont kapunk, ami az adott ponthalmaz konvex burka. A Ponthalmazt feltölthetjük standard inputról is.
A PointSet-en belül definiálva van egy iterátor osztály, iterator ill. const_iterator néven, amivel be tudjuk járni a ponthalmaz pontjait. Az iterátor minden tekintetben a standard iterátoroknak megfelelően viselkedik.
Használat módja:

```
Geometry::PointSet ps, ps1;
std::list<Geometry::Point> pl;
```

```
Geometry::Point p;
for(int i = 0; i < 10; i++){
  std::cin >> p;
  pl.push_back(p);
}
ps.LoadList(pl);
PointSet::iterator it = ps.begin(); PointSet::const_iterator c_it = ps1;
std::cout << it->GetRotate90();
std::cin >> ps1;
```

## Poligon

Megvalósító osztály: ***Poligon***
Síkbeli alakzatokat, poligonokat megvalósító osztály. A PointSet osztállyal ellentétben itt számít, hogy milyen sorrendben kapja a pontokat, amelyek alkotják. A Poligonnak nem lehetnek egymást metsző oldalai, valamint az óramutató járásával ellentétes körbejárás szerint tároljuk a csúcsait.
Szolgáltatások:
Konstruktorok: A default konstruktor egy üres poligont hoz létre. A konstruktornak vagy a LoadList() publikus függvénynek egy std::list<Point> típusú listát megadva is feltölthetjük a poligont. Ekkor a LoadList() ellenőrzi azt, hogy a listában jövő pontok abban a sorrendben megfelelőek-e, nincs-e metsző oldalpár, illetve ha kell megfordítja a sorrendet. A poligont be lehet olvasni standard inputról is.
Műveletek: egy poligonnak meg lehet kérdezni az előjeles területét a SignedArea() függvénnyel. Ez pozitív, ha óramutató járásával ellentétes a pontok listája, negatív, ha megegyező. Az Area() függvénnyel a területet lehet lekérdezni, előjel nélkül. Az IsInside(Point) függvénnyel meg lehet kérdezni egy pontról, hogy az a poligon belsejében van-e. A NodeNum() függvénnyel a poligon csúcsainak számát lehet lekérdezni.
Használat módja:

```
Geometry::Poligon p;
std::cin >> p;
double area = p.Area();
Geometry::Point q;
std::cin >> q;
if(p.IsInside(q)) std::cout << "igen" << std::endl;
```

## Konvex Poligon

Megvalósító osztály: ***Convex***
A konvex poligonokat megvalósító osztály. Jelentősége, hogy könnyen ki lehet

számolni a minimális lefedő téglalapot hozzá.
Szolgáltatások:
Konstruktorok: A default konstruktor egy üres poligont hoz létre. A konstruktornak vagy a LoadList() publikus függvénynek egy std::list<Point> típusú listát megadva is feltölthetjük a konvex poligont. Ekkor a LoadList() ellenőrzi azt, hogy a listában jövő pontok abban a sorrendben megfelelőek-e, nincs-e metsző oldalpár, konvex-e a leírt poligon, illetve ha kell megfordítja a sorrendet. A konvex poligont be lehet olvasni standard inputról is.
Műveletek:
Az örökölt műveleteken kívül egyéb művelet nincs, de ebben az osztályban van benne a minimális lefedő téglalapot kiszámító algoritmus implementációja.
Használat módja:

```
Geometry::Convex c;
std::cin >> c;
```

## Téglalap

Megvalósító osztály: ***Rectangle***
Speciális poligonokat, téglalapokat megvalósító osztály.
Szolgáltatások
Konstruktorok: A default konstruktor egy üres téglalapot hoz létre. A konstruktornak vagy a LoadList() publikus függvénynek egy std::list<Point> típusú listát megadva is feltölthetjük a téglalapot. Ilyenkor a LoadList függvény ellenőrzi, hogy helyes listát adtunk-e meg, azaz 4 pontból áll-e és ezek merőlegesek-e egymásra.
Műveletek: a Poligontól örökli az összes műveletét, az implementációja változik a műveleteknek.
Használat módja:

```
Geometry::Convex c;
Geometry::Rectangle r;
std::cin >> c;
r = c.MinAreaBoundingRect();
```

## Folt

Megvalósító osztály: ***Blob<Image,PixelChecker>***
Egy képen egy foltot (területet) leíró osztály. Két template paramétere van, az elsővel megmondjuk, hogy milyen típusú képen van ez a folt, a második paraméter pedig egy logikát ír le, ami megmondja, hogy egy pixel benne van-e a foltban vagy sem. A folt szerepe az, hogy a szegélyét - azokat a pontokat a képen, amik a folt szélén

vannak – körüljárás szerint sorrendben le tudjuk szedni.
Szolgáltatások:
Konstruktorok: Egy Blob objektum létrehozásához egy képet, egy pontot a foltban és egy PixelChecker objektumot kell átadni. A PixelChecker zárójel operátorának a kép adott pixelén igazat kell adnia – vagyis olyan pozíciót kell megadni, ami a folt egy pontja.
Műveletek: A GetBorder() függvény segítségével a folt szegélyét kapjuk meg.

### Szegély

Megvalósító osztály: ***Border***
Egy raszteres objektum szegmenst határait tartalmazó pontsorozatot nevezünk szegélynek. Azért hasznos, mert ez természetes átmenetet képez az objektum raszteres és vektoros reprezentációja közt.
Szolgáltatások:
Konstruktorok: A default konstruktor egy üres szegélyt hoz létre. A konstruktort egy std::list<Point> típusú listával paraméterezve is feltölthetjük a szegélyt.
Műveletek: A szegély pontjait el tudjuk szögletes zárójelles indexeléssel. A Poligonize() függvény egy rekurzív poligonizáló algoritmus, melynek eredménye egy poligon, melyre igaz, hogy nincs 3 egymás utáni p, q, r pontja, amelyekre teljesül, hogy Distance(Line(p,r),q) < limit, ahol a limit egy globális változó a Geometry névtérben.
Használat módja:

```
struct MyChecker{
  bool operator()(const Pixel<3,8>& pix) const{
    return (pix[0] > 128 && pix[1] > 128 && pix[2] > 128);
  }
};
Image<3,8> img(200,200);
MyLoadImage(img,"whiterabbit_in_center.bmp")
Geometry::Blob<Image<3,8>,MyChecker>  blob(img,Position(100,100));
Geometry::Border border = blob.GetBorder();
Geometry::Poligon p = border.Poligonize();
```

## Geometriai függvények

A Geometry névtérben sok hasznos függvény is helyet kapott, amelyek segítségével gyakran előforduló kérdésekre tudunk választ kapni.

Két vektor által közbezárt szög
*double Angle(const Vector p, const Vector q)*
A p és q vektorok által közbezárt szöget adja vissza radiánban.

**Két egyenes metszéspontja**
*Point Cut(const Line& e, const Line& f)*
Egy Point típusú objektumban visszaadja az e és f egyenesek metszéspontját (ha létezik).

**Két szakasz metszéspontja (ha létezik)**
*Point Cut(const Segment& s, const Segment& t)*
Egy Point típusú objektumban visszaadja az e és f egyenesek metszéspontját (ha létezik).

**Pont és egyenes előjeles távolsága**
*double SignedDistance(const Line l, const Point p)*
Az l egyenes és p pont előjeles távolságát adja meg. Az előjel mondja meg, hogy a pont az l jobb-, vagy bal oldalára esik.

**Pont és egyenes távolsága**
*double Distance(const Line l, const Point p)*
Az l egyenes és p pont távolságát mondja meg.

**Két pont távolsága**
*double Distance(const Point p, const Point q)*
A p és q pontok távolságát adja meg.

**Két szakasz metszi-e egymást**
*bool ExistsCut(const Segment& s, const Segment& t)*
Megmondja, hogy az s és t szakaszok metszik-e egymást.

**Két egyenes szögfelezője**
*Line Median(const Line e, const Line f)*
Az e és f egyenesek szögfelező egyenesét adja vissza egy Line típusú objektumban.

**Két egyenes párhuzamos-e**
*bool Parallel(const Line& e, const Line& f)*
Egy logikai értékben visszaadja, hogy az e és f egyenesek párhuzamosak-e.

**Pont projekciója egy egyenesre**
*Point Projection(const Line& e, const Point& p)*
Egy Point típusú objektumban visszaadja a p pont e egyenesre eső merőleges vetületét.

## Szintézis

Névtér: MyDraw

A szintézist támogató eljárások célja, hogy a Geometry névtérben definiált objektumokat ki tudjuk rajzolni képekre. Hogy a generált képek szemléletesebbek legyenek, szükség lehet arra, hogy pontokat, vonalakat többféle stílussal is ki lehessen rajzolni. Ezt templatek használatával értem el. Az minden egyes kirajzoló osztályban egyetlen kirajzoló függvény van, aminek a neve Draw. Ennek a paraméterezése a következő konvenció szerint történik: első paramétere a kép amire ki kell rajzolni az objektumot, második maga a kirajzolandó objektum, a harmadik pedig egy a képnek megfelelő Pixel objektum, ami megmondja, hogy milyen színnel kell kirajzolni. Minden osztálynak template paramétere a képtípus, amire rajzolni kell.

***class PointDrawer<Image>***
void Draw(Image& img, const Geometry::Point& point, const Pixel<Image::spp,Image::bps> pix)
Kirajzol egy pontot, összesen 9 pixelre: a ponthoz legközelebbi pixelre és annak 8 szomszédjára.
***class PointPureDrawer<Image>***
void Draw(Image& img, const Geometry::Point& point, const Pixel<Image::spp,Image::bps> pix)
Kirajzol egy pontot: a ponthoz legközelebbi pixelt állítja a kívánt színre.
***class P2PSimpleDrawer<Image>***
void Draw(Image& img, const Geometry::Segment& s, const Pixel<Image::spp,Image::bps> pix)
Kirajzolja egy szakasz két végpontját összekötő egyenest. A két végpontot nem rajzolja ki.
***class P2PDottedDrawer<Image>***
void Draw(Image& img, const Geometry::Segment& s, const Pixel<Image::spp,Image::bps> pix)
Kirajzolja egy szakasz két végpontját összekötő egyenest, szaggatott vonallal. A két végpontot nem rajzolja ki.
***class SegmentDrawer<Image, PDrawer, P2PDrawer>***
void Draw(Image& img, const Geometry::Segment& s, const Pixel<Image::spp,Image::bps> pix)
A template paraméterben kapott PDrawer típusú kirajzolóval kirajzolja egy szakasz két végpontját, az összekötő egyenest pedig a kapott P2PDrawer típusú kirajzolóval.
***class PoligonDrawer<Image, PDrawer, P2PDrawer>***
void Draw(Image& img, const Geometry::Poligon& p, const Pixel<Image::spp,Image::bps> pix)
A template paraméterben kapott PDrawer típusú kirajzolóval kirajzolja egy poligon pontjait, az éleket pedig a kapott P2PDrawer típusú kirajzolóval.
***class PointSetDrawer<Image, PDrawer>***
void Draw(Image& img, const Geometry::PointSet& p, const Pixel<Image::spp,Image::bps> pix)
A template paraméterben kapott PDrawer típusú kirajzolóval kirajzolja egy ponthalmaz pontjait.

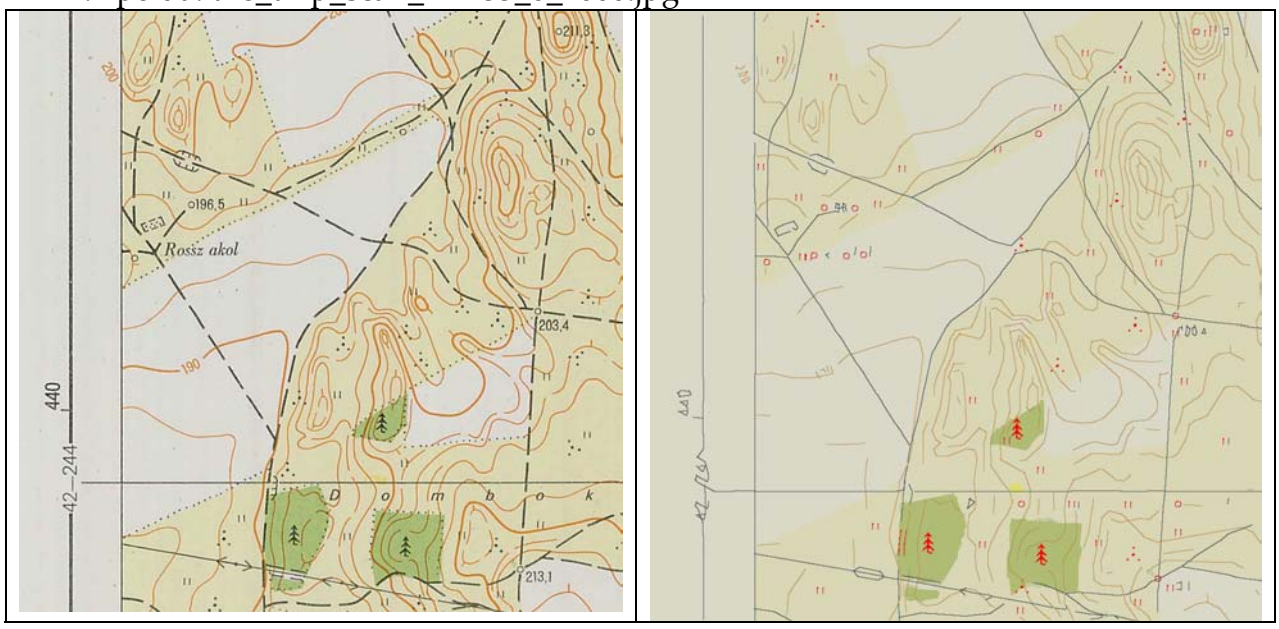
Használat módja:

```
Image<3,8> img(400,400);
Pixel<3,8> pix; pix[0] = 255; pix[1] = 0; pix[2] = 0;
Geometry::Point p(100,100), q(200,200);
MyDraw::PointDrawer<Image<3,8> > pd;
pd.Draw(img,p,pix);
MyDraw::SegmentDrawer<Image<3,8>,   MyDraw::PointPureDrawer<Image<3,8>   >,
MyDraw::P2PDottedDrawer<Image<3,8> > > sd;
//az sd szakaszokat rajzol ki, sima végponttal és szaggatott vonallal
MyDraw::PoligonDrawer<Image<3,8>,       MyDraw::PointDrawer<Image<3,8>       >,
MyDraw::P2PSimpleDrawer<Image<3,8> > > pd;
//a pd poligonokat rajzol ki, sima vonallal és kiemelt csúcsokkal
sd.Draw(img,Geometry::Segment(p,q),pix);
Geometry::Poligon poligon; std::cin >> poligon;
pd.Draw(img,poligon,pix);
```

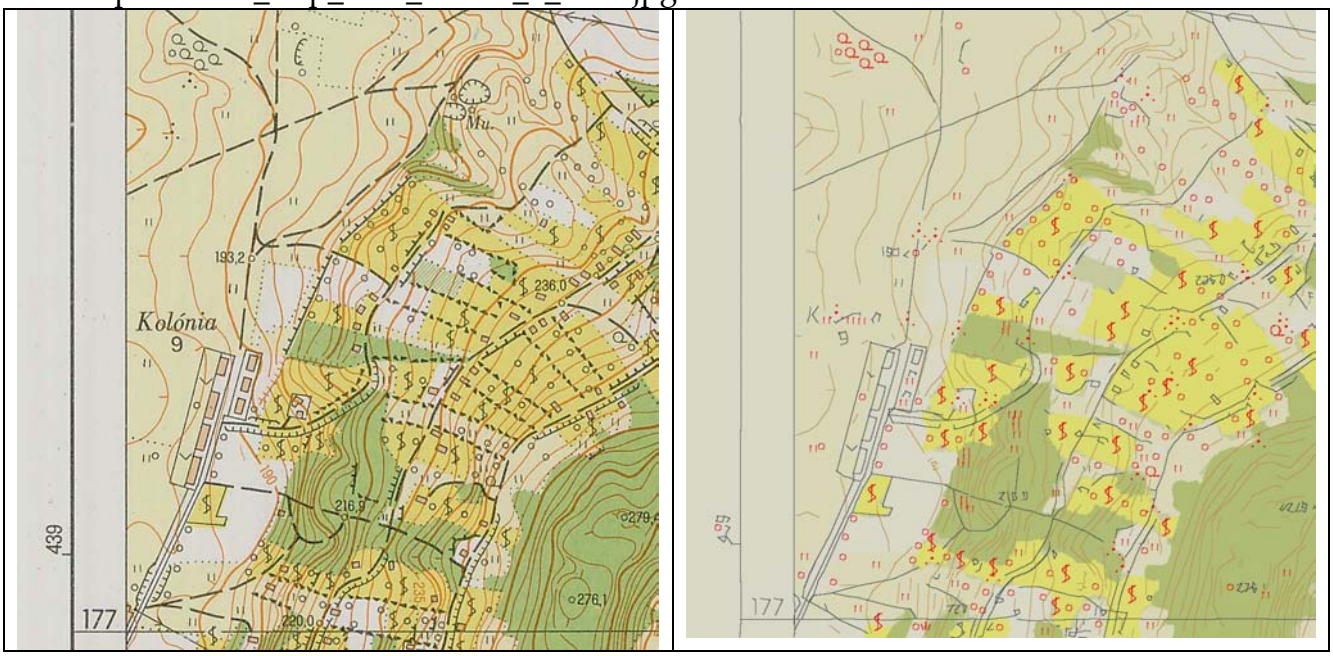
# 7. fejezet: Gyakorlati példák

Ebben a fejezetben gyakorlati példákon mutatjuk be az IRIS rendszer működését. A különböző térképfajták alapvetően eltérő természetű geometriai objektumokkal dolgoznak, így az eljárások bizonyos térképfajtákra lettek „kihegyezve”. Mivel az IRIS rendszert a magyarországi topográfiai térképek vektorizálására optimalizáltuk, ezért a rendszer ezekre működik megfelelően.
A projekt kezdeti szakaszában úgy terveztük, hogy a fejlesztés folyamán elkészülő eljárások processz folyamokba, ugynevezett workflowkba szervezve fognak működni, amelyeket a végfelhasználók kedvük, tudásuk, tapasztalataik szerint állíthatnak össze. Ez a koncepció meg is maradt, de a kezdeti tervekhez képes annyiban változott, hogy a végfelhasználó általi tetszőleges workflow összeállítást nem tettük lehetővé, mivel úgy találtuk, hogy egy jól működő workflow összeállítása olyan mértékben bonyolult feladat, hogy azt egy mégoly képzett végfelhasználó sem fogja tudni összeállítani. Mi magunk természetesen összeállítottunk a magyar topográfiai térképeket megfelelő minőségben vektorizáló workflowt, ami végül is az IRIS program üzleti logikáját adja.
A térképészeti konvenciók országról országra különböznek, a színvilág, az alkalmazott jelkulcs mind eltérhet egymástól. Ezért aztán nem meglepő, ha más országbeli térképekre a rendszer első futása nem ad kielégítő eredményt. Az eddigi tapasztalatok azonban azt mutatják, hogy a jelkulcs megtanítható a rendszernek, a wokflowk finoman összeállíthatók a fejlesztők által.
A következőkben bemutatunk néhány raszteres térképet, és a vektorizálásként kapott vektoros shape fájlokat, amelyeket az ESRI ArcGIS 9.1 szoftverével jelenítettünk meg. A bal oldali kép raszteres, a jobb oldali vektoros.

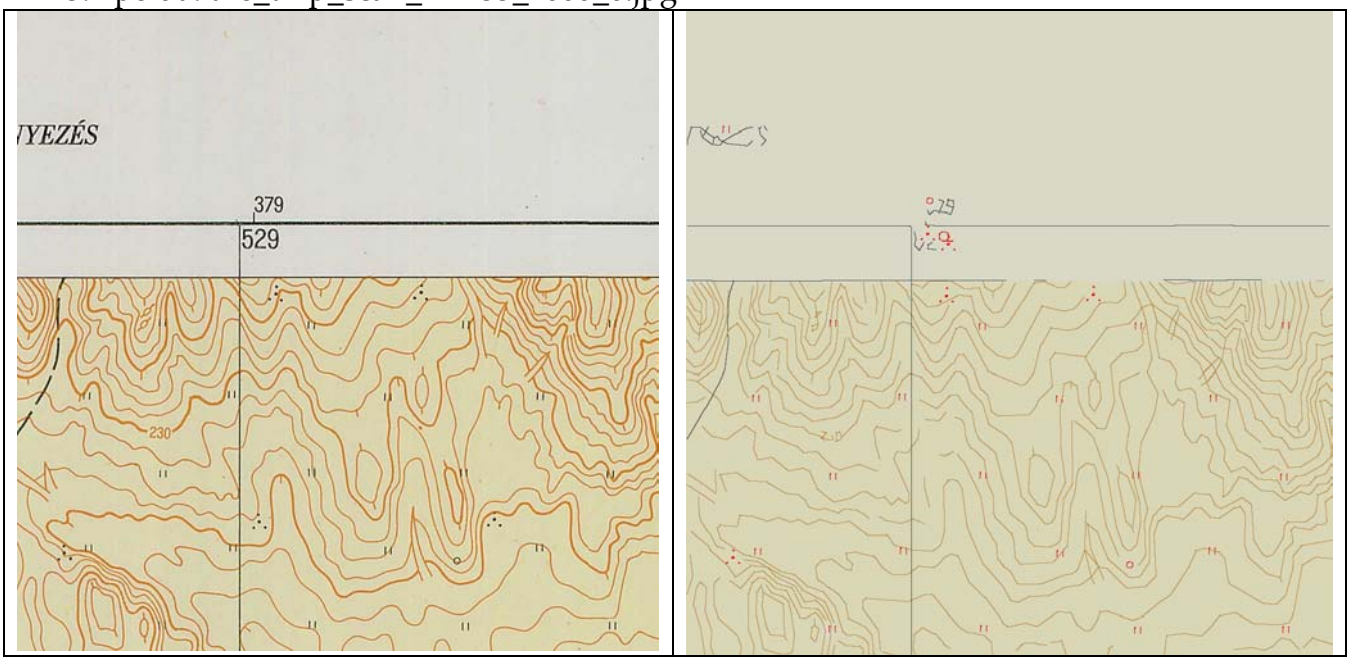
1. példa: tile_tmp_scan_42-133_0_2000.jpg



2. példa: tile_tmp_scan_42-133_0_3000.jpg

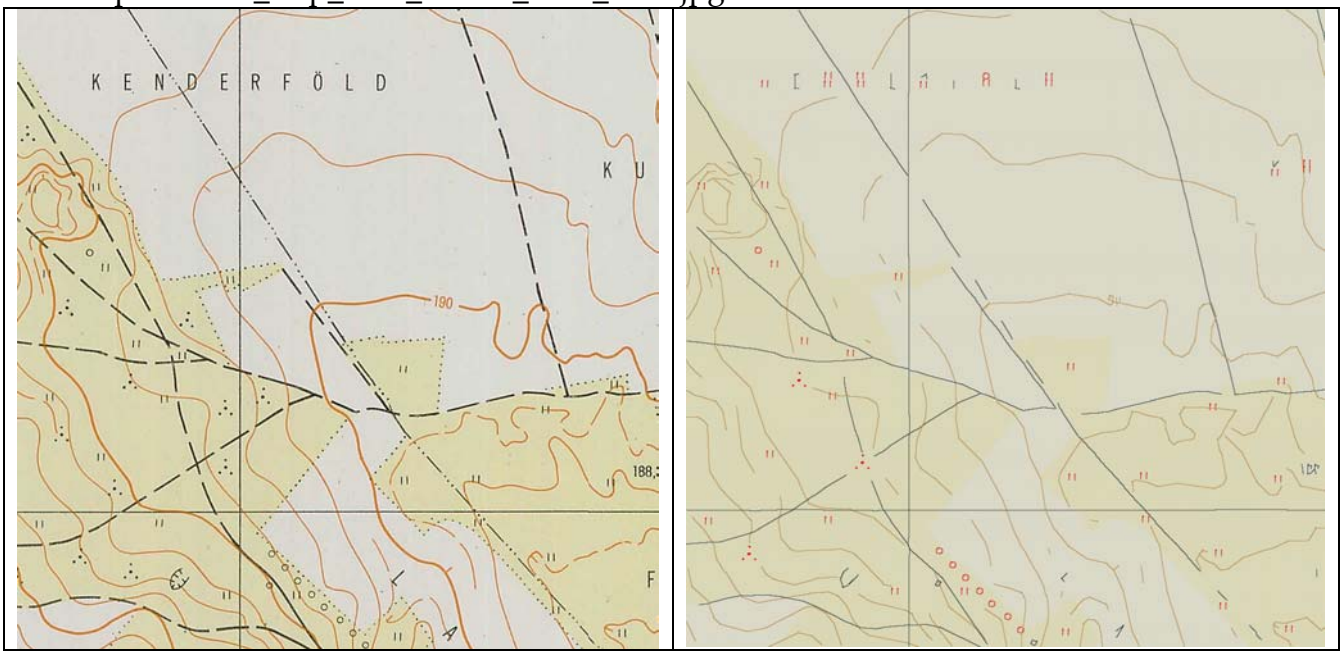



3. példa: tile_tmp_scan_42-133_1000_0.jpg

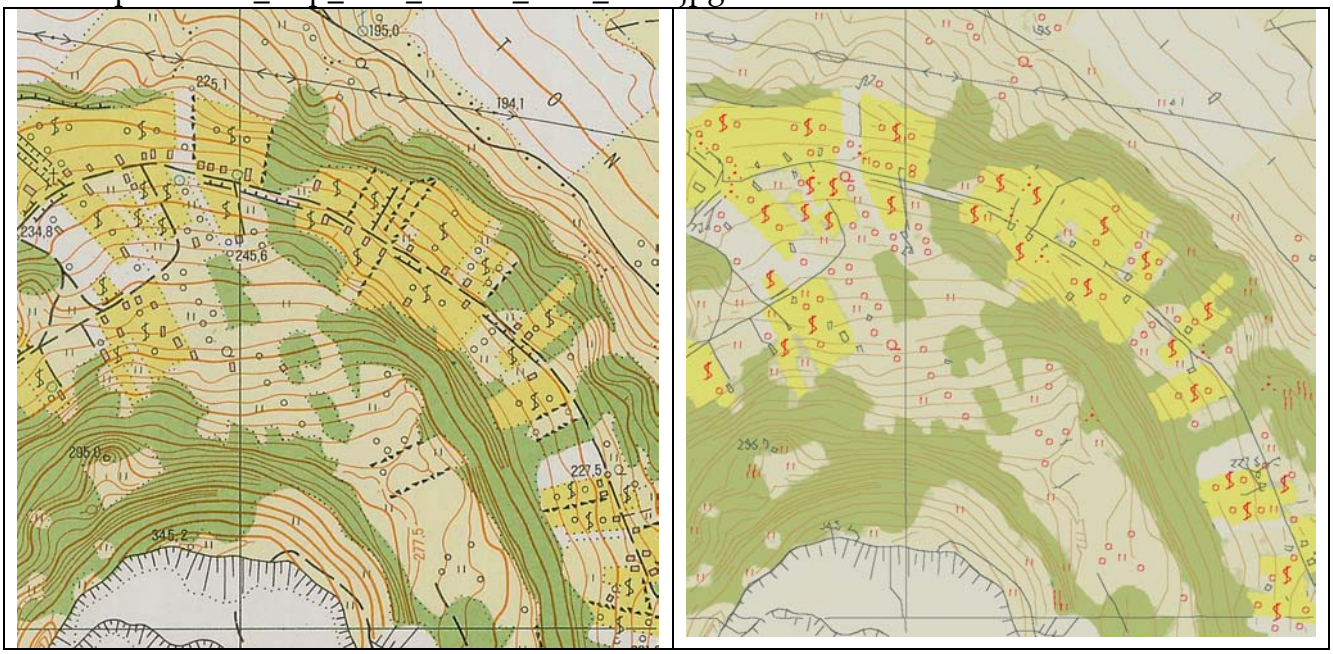


4. példa: tile_tmp_scan_42-133_1000_1000.jpg

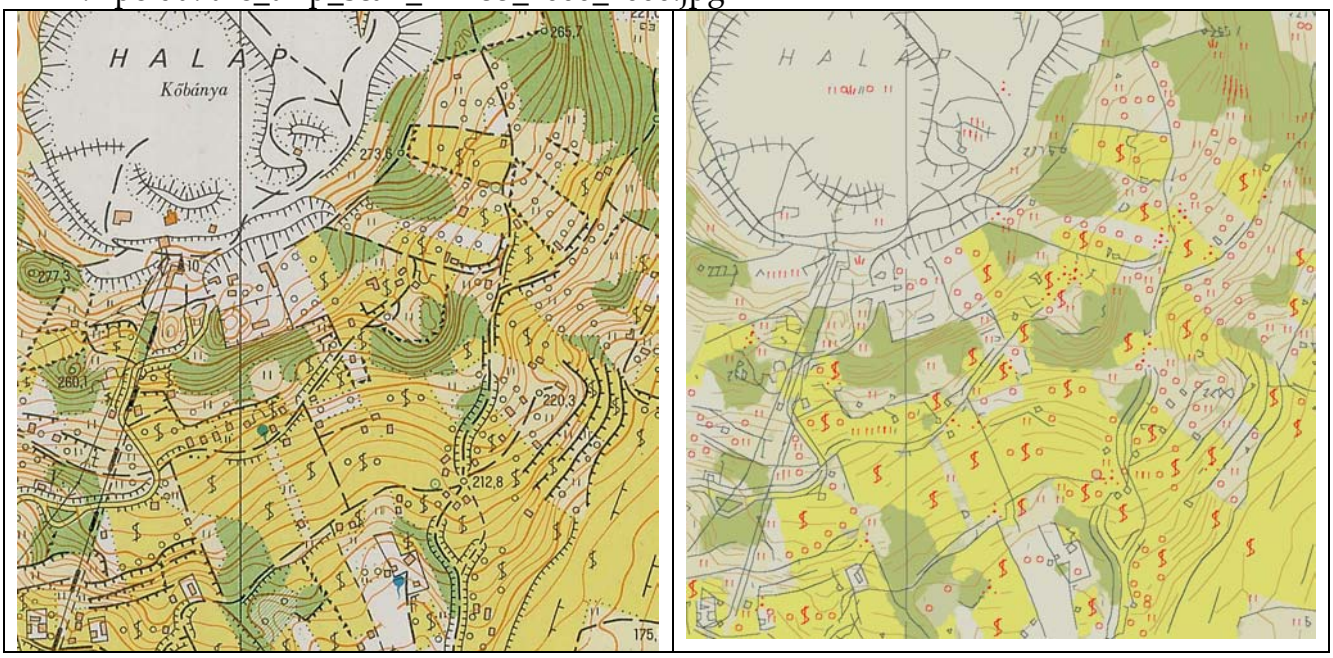

5. példa: tile_tmp_scan_42-133_1000_2000.jpg



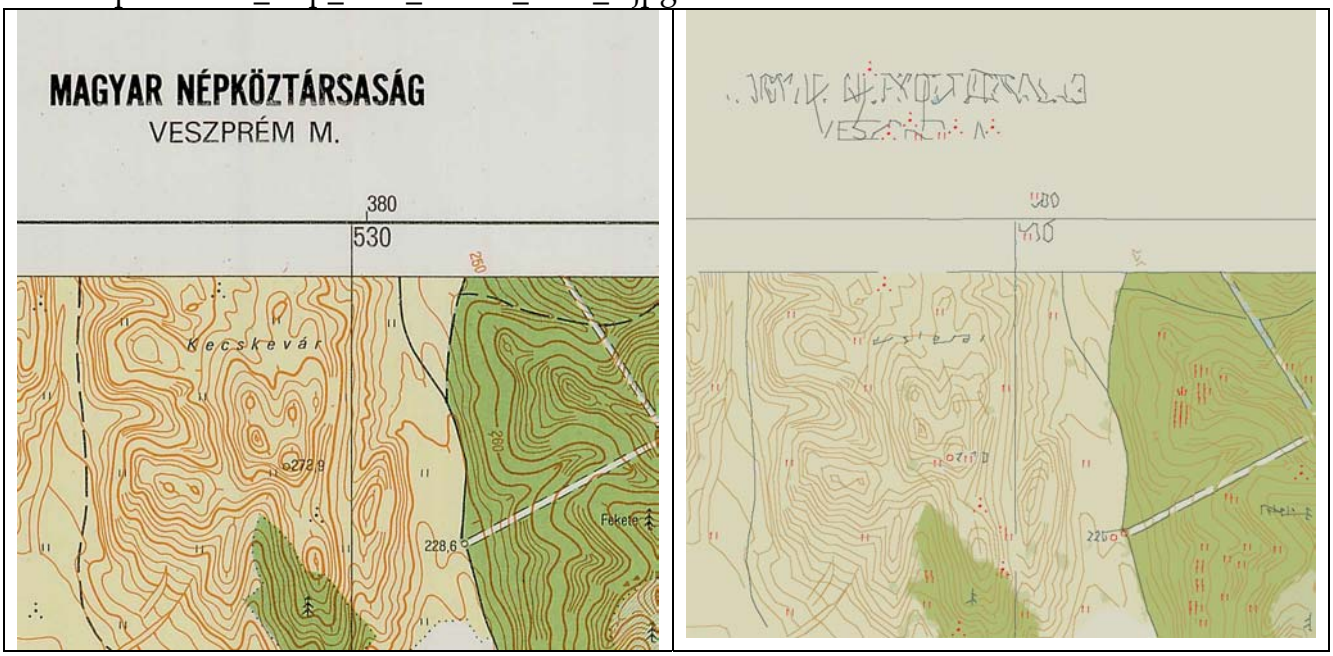
6. példa: tile_tmp_scan_42-133_1000_3000.jpg



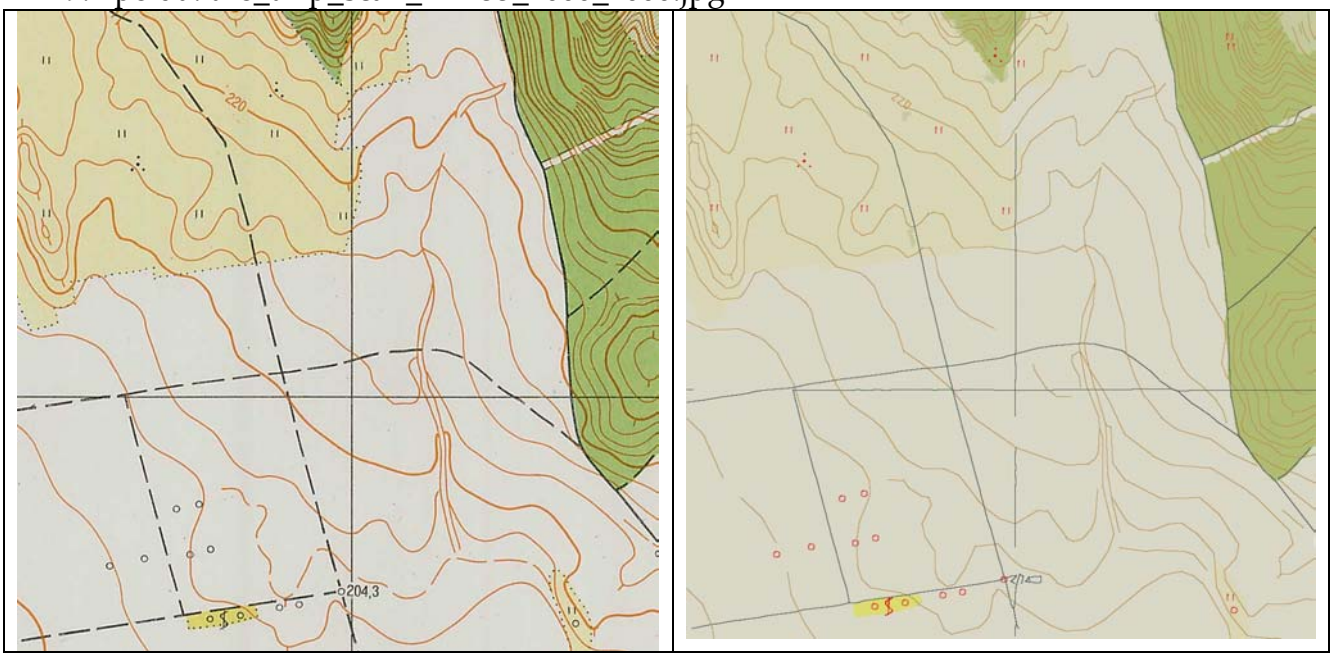

7. példa: tile_tmp_scan_42-133_1000_4000.jpg



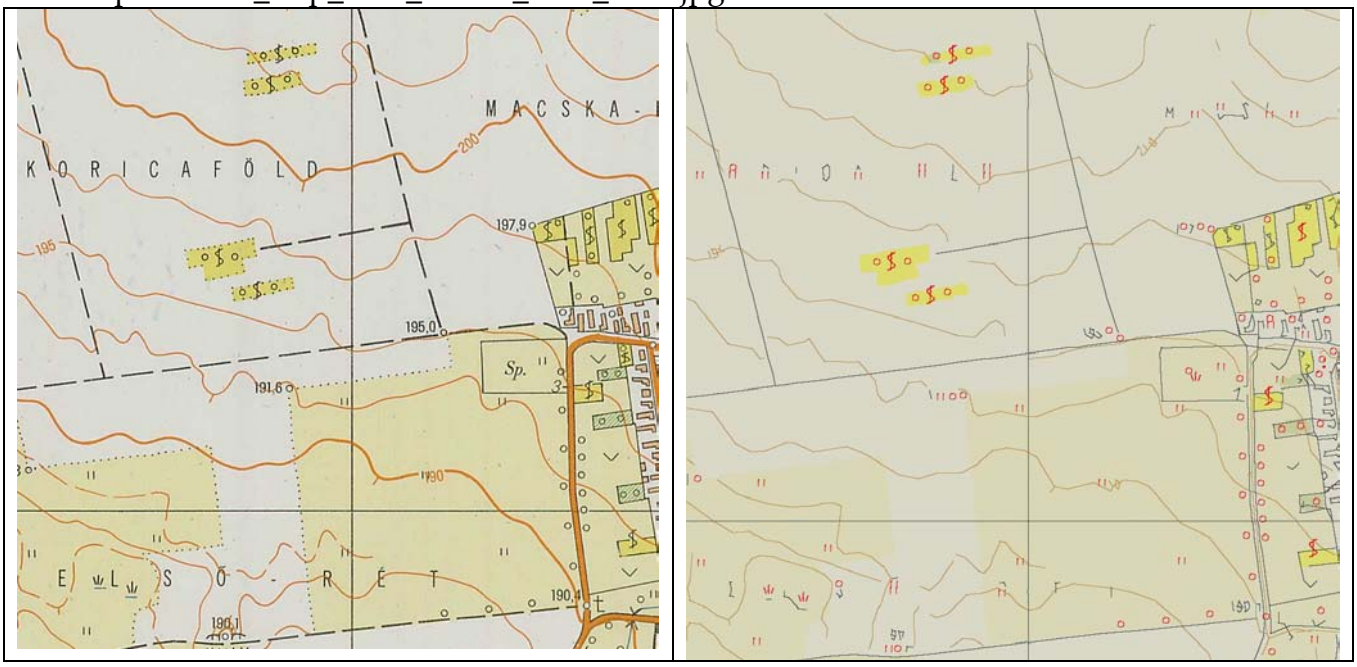
8. példa: tile_tmp_scan_42-133_2000_0.jpg



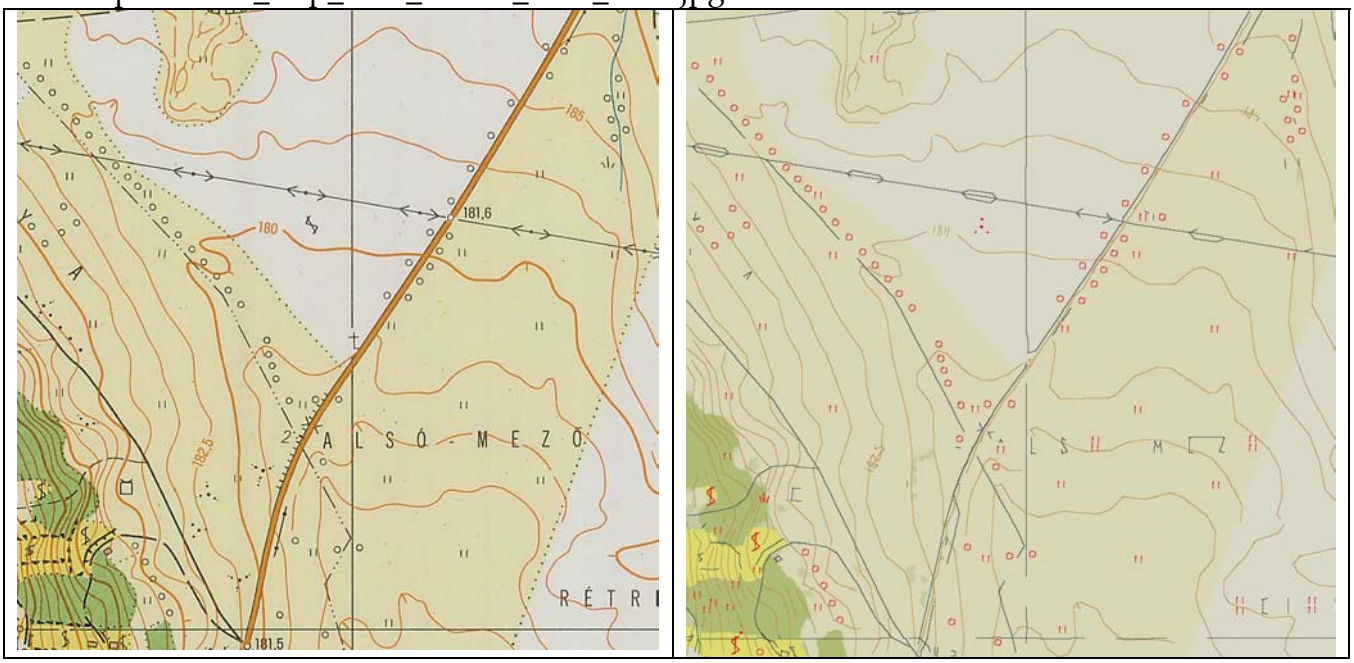

9. példa: tile_tmp_scan_42-133_2000_1000.jpg



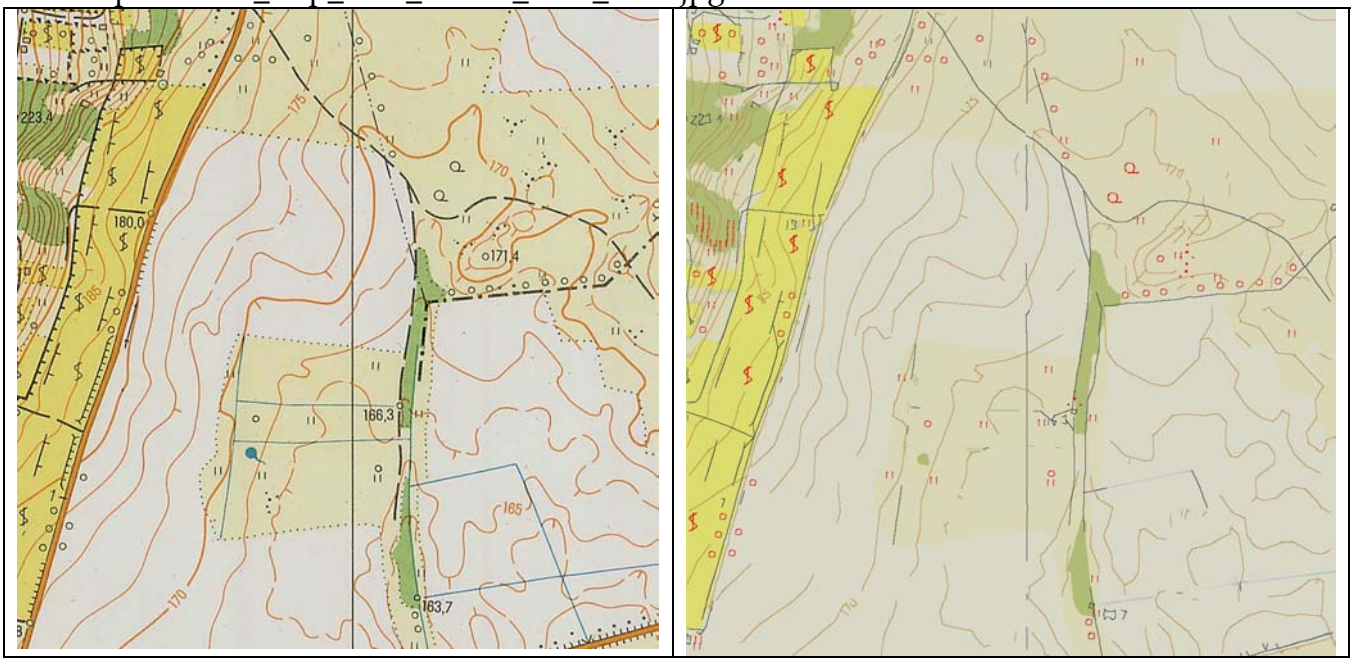
10. példa: tile_tmp_scan_42-133_2000_2000.jpg




11. példa: tile_tmp_scan_42-133_2000_3000.jpg

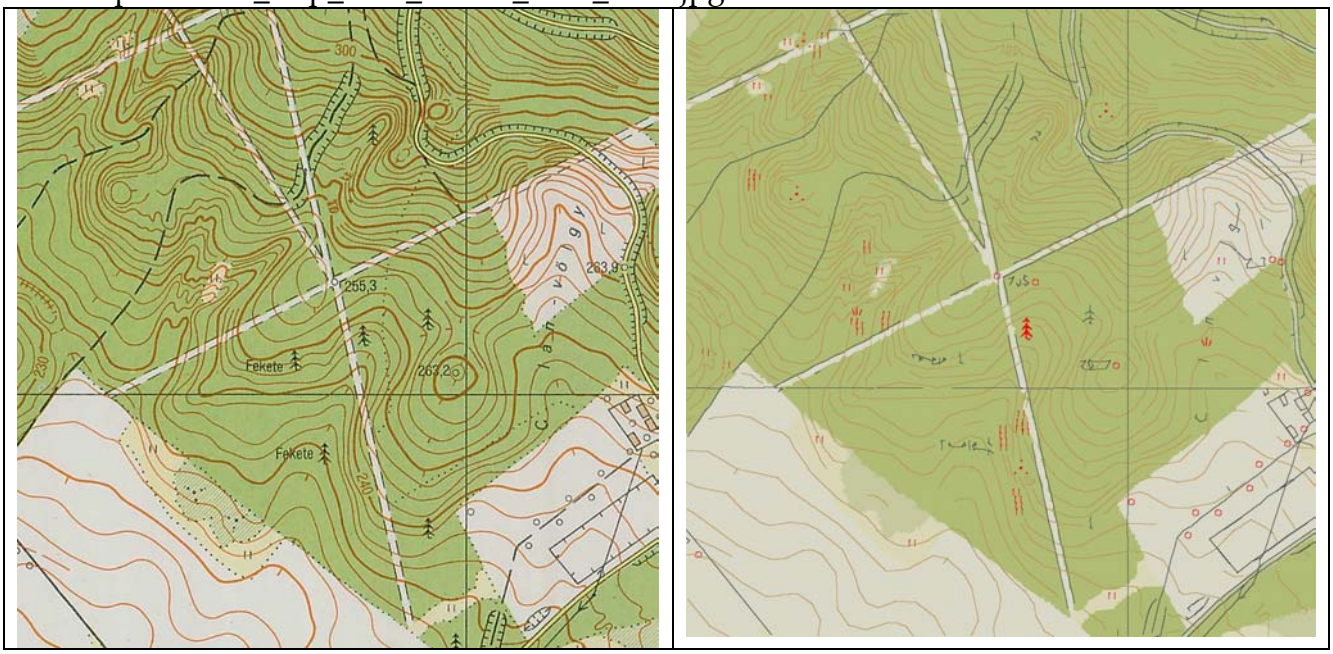


12. példa: tile_tmp_scan_42-133_2000_4000.jpg

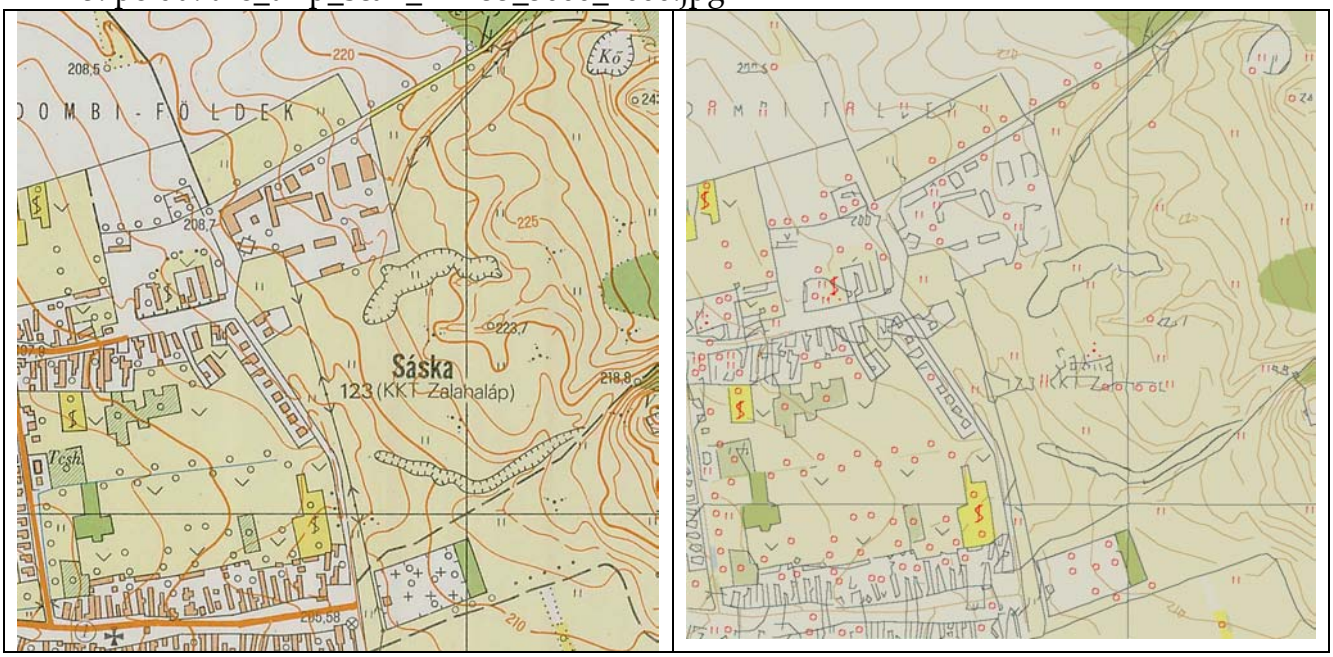


13. példa: tile_tmp_scan_42-133_3000_0.jpg

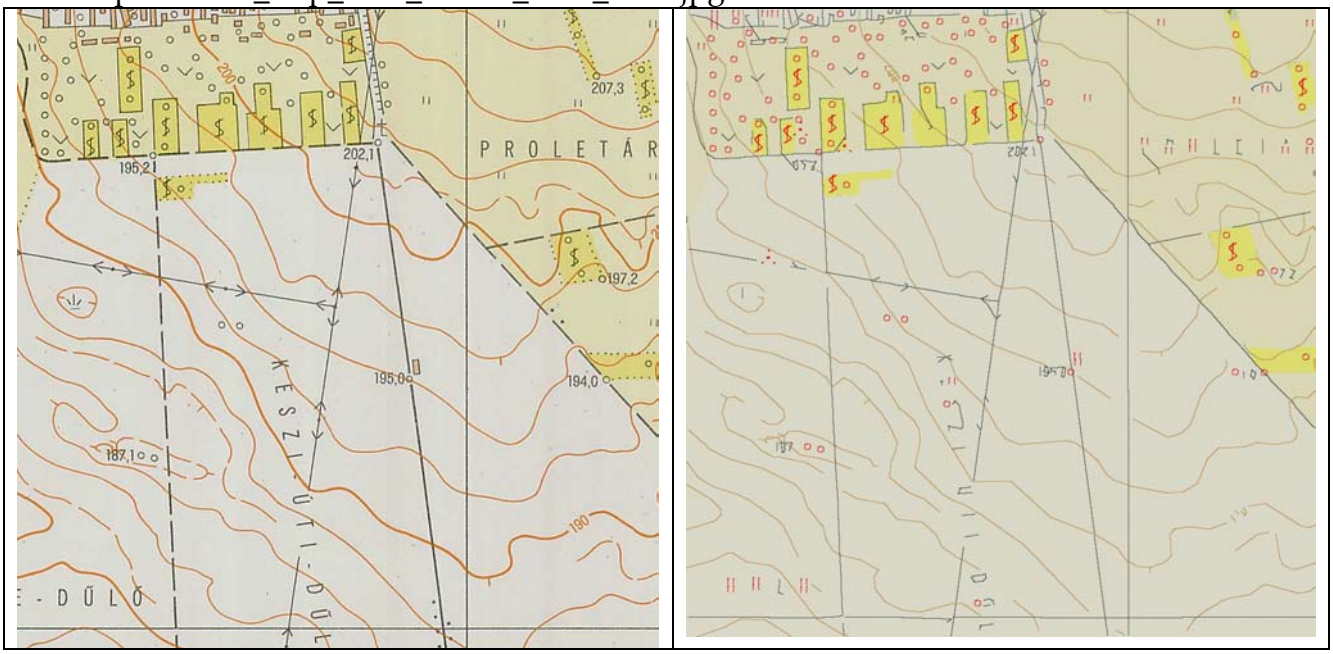
14. példa: tile_tmp_scan_42-133_3000_1000.jpg



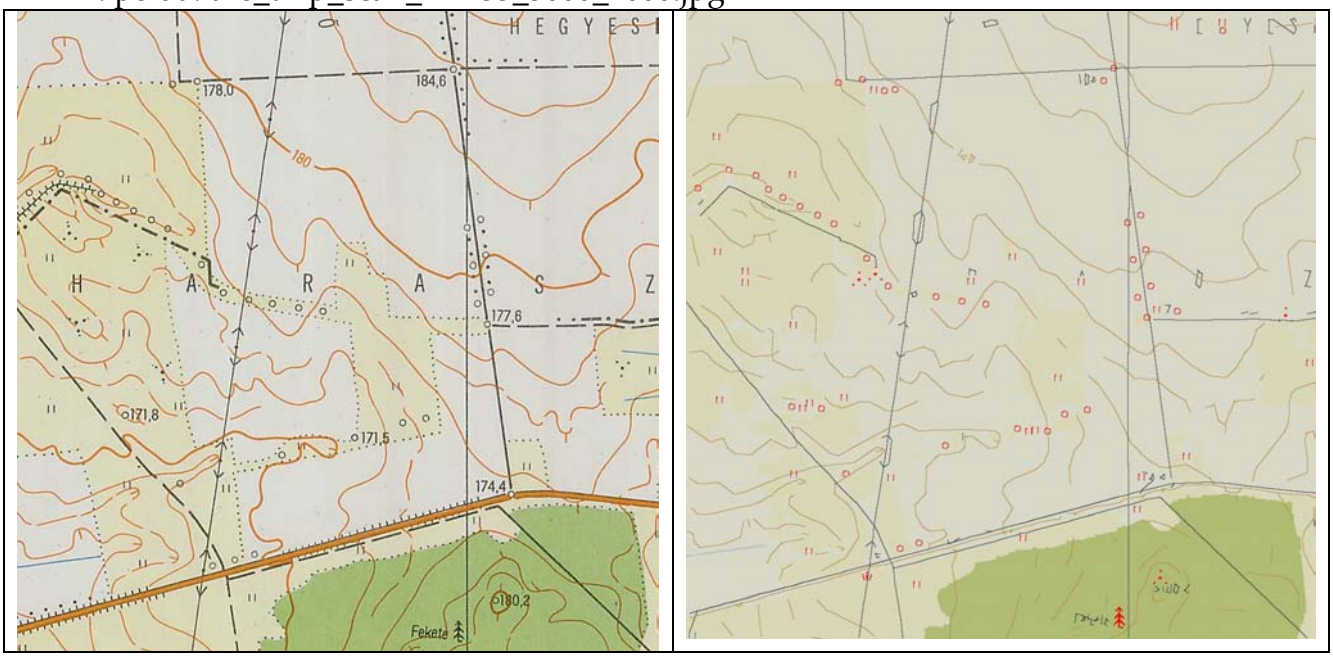

15. példa: tile_tmp_scan_42-133_3000_2000.jpg



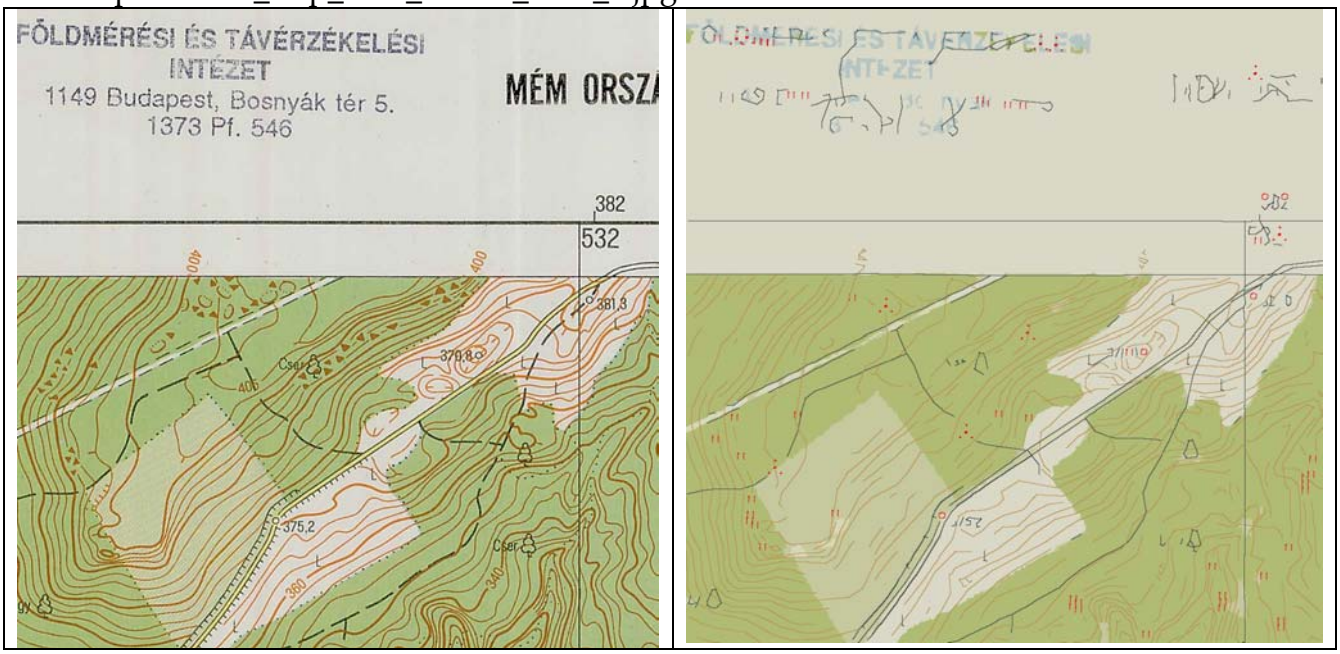
16. példa: tile_tmp_scan_42-133_3000_3000.jpg



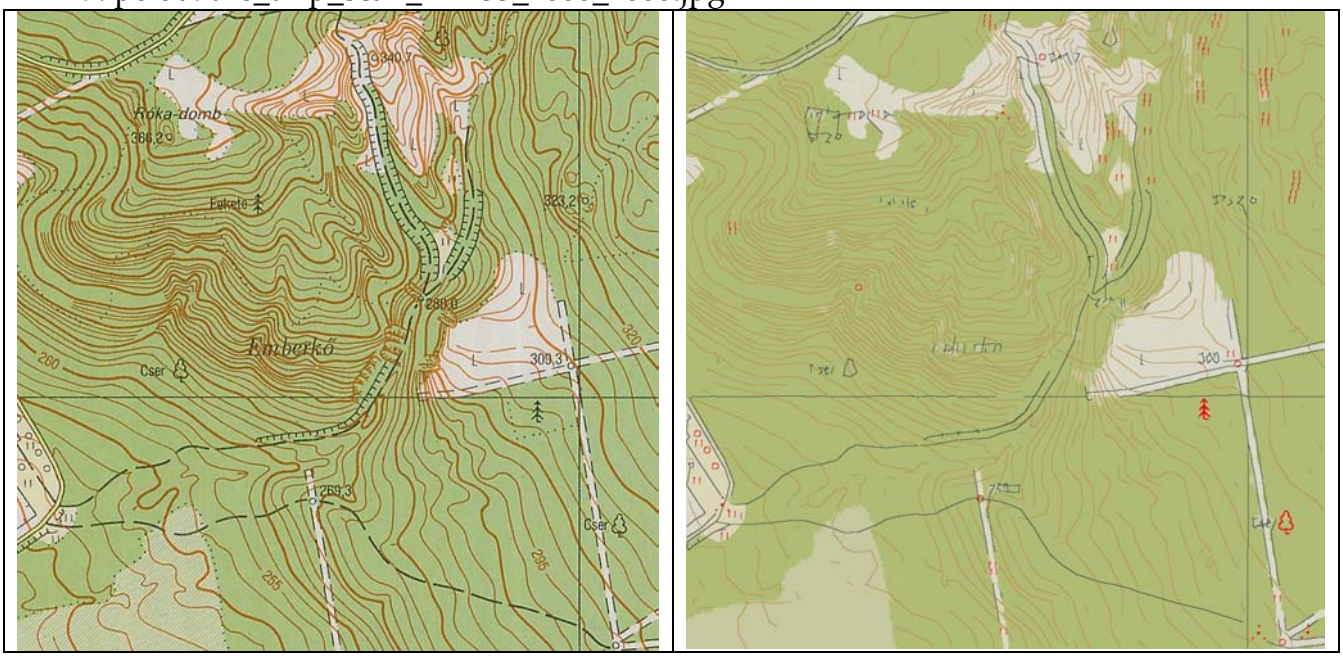

17. példa: tile_tmp_scan_42-133_3000_4000.jpg



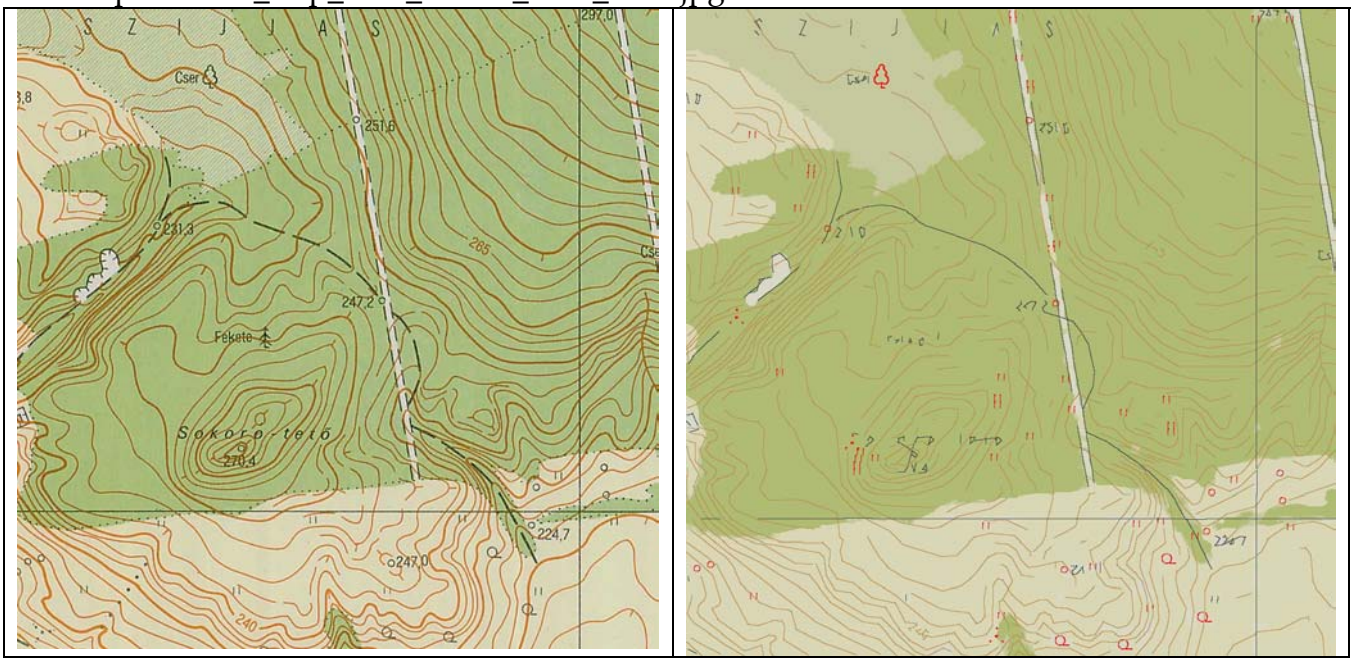
18. példa: tile_tmp_scan_42-133_4000_0.jpg



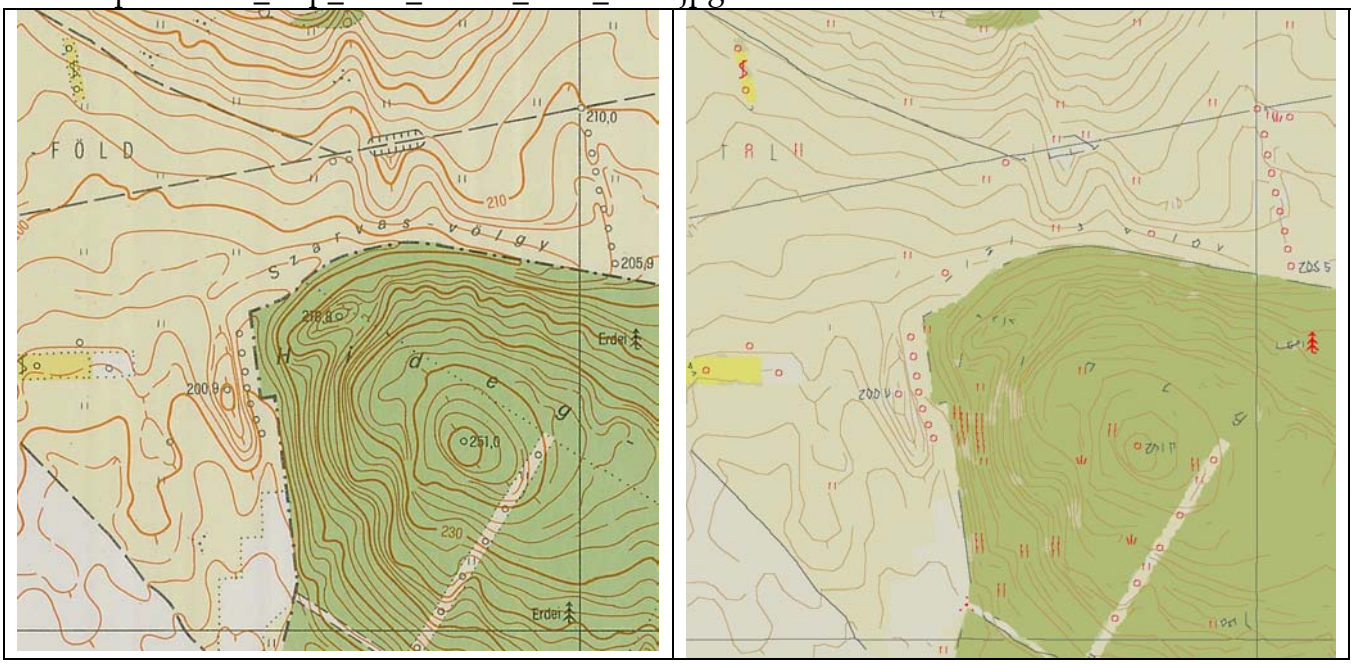

19. példa: tile_tmp_scan_42-133_4000_1000.jpg



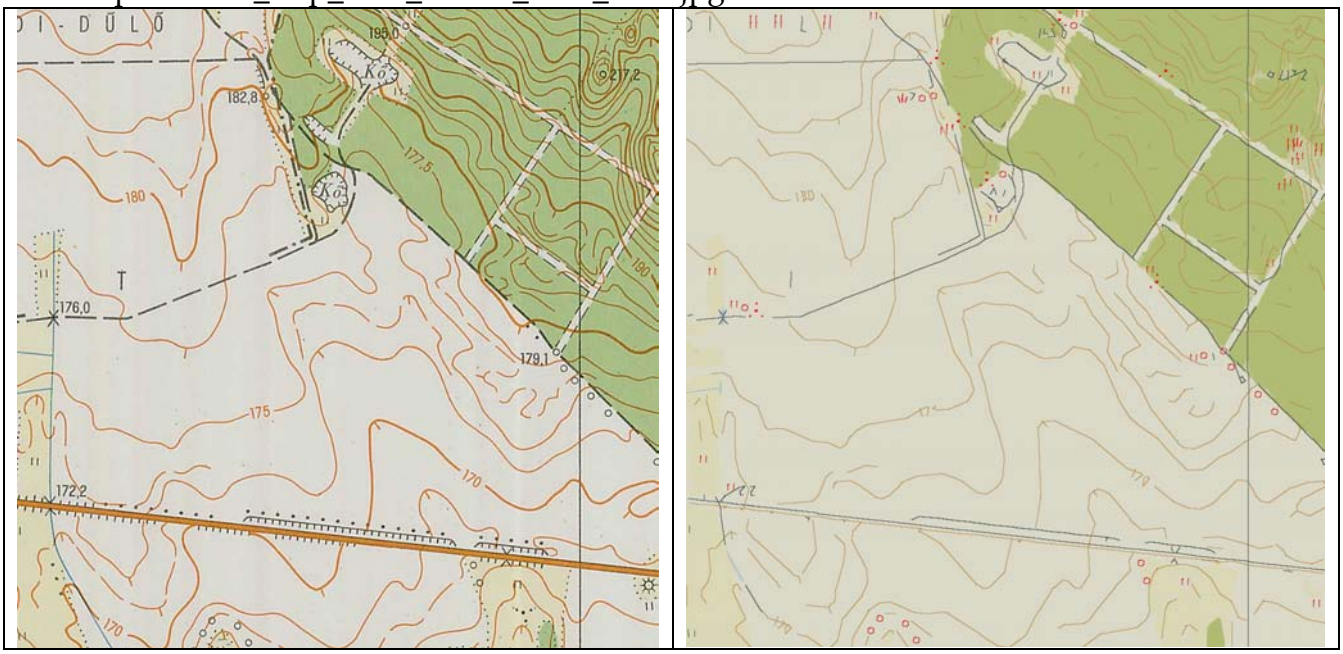
20. példa: tile_tmp_scan_42-133_4000_2000.jpg




21. példa: tile_tmp_scan_42-133_4000_3000.jpg



22. példa: tile_tmp_scan_42-133_4000_4000.jpg



# 8. fejezet: Hogyan tovább

## *Piaci verzió előállítása*

Az IRIS projekt futamidejének három éve alatt két alapvető eredményt sikerült elérni. Az egyik az, hogy sikerült jól működő vektorizáló szoftvert kifejleszteni. A másik eredményt az, hogy világossá vált számunkra, a mesterséges intelligencia ma létező módszereivel nem tudjuk az ember térképészeti tudását átadni egy computernek. Ügyes algoritmusokkal, jól összeállított workflowkkal persze egyre jobban működő eljárásokat lehet megalkotni, de ezek önmaguktól sosem fognak fejlődni, új ismeretet önerőből sosem fognak szerezni. Fejlődésük egyetlen lehetséges módja a valóban intelligens emberi közreműködő beavatkozása a program működésébe új ismeret elemekkel, új eljárásokkal, új modulok megalkotásával.
Az elsőnek említett eredmény gyakorlati bizonyítékait (és egyben hiányosságait is) az előző fejezetben láthattuk. Összetett algoritmusokkal, szűrési módszerekkel elég jó minőségben előállíthatók vektoros adatok raszteres térképi állományokból. Az eljárások semmiféle gépi intelligenciát, tudásbázis nem alkalmaznak. Helyes működésük abban rejlik, hogy az alkotók saját intelligenciájukat építették be az eljárásokba. Ami tudás talán mégis benne van, az a jelkulcs mintázatának tárolása, amely semmilyen ismérv alapján sem nevezhető intelligenciának vagy tudásbázisnak.
A hogyan tovább kérdésre tehát az egyik lehetséges válasz, hogy haladjunk az eddig működőképesnek látszó úton, amely saját tudásunkra épített algoritmusok beépítésével egyre kifinomultabb eredményeket fog szolgáltatni. Még az is előfordulhat, hogy egy jól belőtt térképtipusra akár tökéletes eredményt produkáljon, vagyis piaci szoftververzió kidolgozása is értelmes célkitűzés lehet.

## *Az IRIS rendszer alapkutatási vonatkozásai*

A másik lehetőség egy rögösebb út választása lehet, amelynek során létrehozunk egy tudásbázist, amelynek célja a számítógép emberhez hasonló működése, mármint egy térképet olvasó emberé. Ehhez azonban számos alapkérdés tisztázásra vár. A következő, és egyben utolsó fejezetben ismertetjük egy általunk felvázolt, úgynevezett Digitális Evolúciós Gép (DEM: Digital Evolution Machine) működésének elvi alapjait. Tényleges kipróbálása a projekt folytatása esetén volna lehetséges.
Ebben a fejezetbenben megpróbáljuk felvázolni azokat az alapelveket, amelyek egy elképzelt digitális lény „gondolkozásának" kialakulását szabályozzák. A „gondolkodás" kialakulásának ott van szerepe, hogy az elképzelt lény képes legyen a

saját maga boldogulását biztosító működésre. Képes legyen leképezni a környezet tulajdonságait, ezen tulajdonságokat saját ,,érdekeivel" egyeztetni, összevetni, saját működési módját, harcmodorát a környezethez igazítani. Ha mindezt sikerül megtennie, akkor megfelelő modellje lehet egy önfejlődésre képes rendszernek.
A mesterséges intelligenciának nevezett tudományterület eddigi eredményei óriásiak, hiszen időnként fantasztikus képességű rendszerek születnek, azonban a létező rendszerekből mégsem következik az a jövőkép, hogy a digitális gondolkodás felvegye a versenyt akár a legegyszerűbb emlős állatéval is. Ez a megállapítás nem annyira egy adott működés minőségére, mint inkább annak fejlődőképességére értendő. Számos célrendszer intelligensnek látszó képességeket mutat, de ezen működés érvényességi köre meglehetősen korlátozott, sőt e határok alig bővíthetők külső beavatkozás nélkül.
Messziről indítanám a probléma feltárását. Vegyünk egy egyszerű, fiktív esetet. Sodródik az ősóceánban egy valamilyen meglehetősen primitív, ősi életformájú lény. Azt veszi észre, hogy a víz túl hideg kezd lenni a számára. Kiértékeli a környezetét. Összehasonlítja a mért hőmérsékletet a számára kedvező ideálissal. Ha nagy az eltérés, dönt, ,,elúszom innen'' (1. ábra).

1. ábra
A lények evolúciójára a legnagyobb hatást az időjárás, és a klíma változásai tették. A mindenkori környezethez való alkalmazkodás kényszere, és a többi lény létezéséből származó problémák kezelése kellett, hogy kiváltsa egy a környezet ,,megértésére'', vagy legalábbis az események megjegyzésére és felidézésére alkalmas tudásbázis kialakulását.}

Ez vajon már intelligencia, hiszen észlel, kiértékel, dönt, cselekszik? Kézenfekvőnek tűnik a válasz, hogy igen. Intelligensnek tűnik a viselkedése. Ami tehát észlel, kiértékel, dönt és cselekszik az már intelligens? Nem, mert ha ez igaz lenne, akkor egy termosztát is intelligens volna. Joggal merülhet fel a kérdés, hogy miért nem intelligensek a mérő, kiértékelő, döntő és cselekvő robotok? Ha képesek volnának okulni a tapasztalataikból, akkor már intelligensek lennének? Az okulás a tudásbázis

folyamatos fejlődését jelenti, de vajon mikor lépjük át azt a határt, amelyen túl valóban intelligensnek fogjuk találni a lényt? Az okulásnak bizonyos tudásmennyiség után a tudásbázis szerkezeti átalakulását, és a meglévő tudás átértékelését is kell okoznia ahhoz, hogy átlépjük a linearitásból eredő túl egyszerű gondolkodást.
A következőkben összefoglalom a tudásbázisra és annak működésére vonatkozó alapelveket, a legfontosabb szerkezeti megállapításokat és a működésre vonatkozó legfontosabb lépéseket.

## A tudásbázis alapelvei

Mivel még nem tudjuk pontosan, hogy milyen lesz szerkezetileg és működését tekintve a tudásbázis, ezért fogalmazzuk meg fenomenológikusan, felszínes ismérvek szerint, a mély struktúrák ismerete nélkül, hogy milyen legyen a majdani tudásbázis.

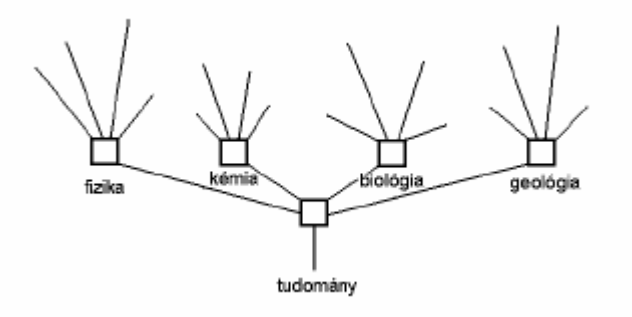
### Általános alapelvek

- A tudásbázis tudáselemekből áll, legyenek ezek az atomok.
- Valamennyi atom kapcsolódik legalább egy másik atomhoz.
- A már egyszer bejárt útvonalak legyenek jutalmazva, annál inkább, minél többször mentek végig rajtuk (a kapcsolat megerősítése). Ez egyben azt is jelenti, hogy a ritkán használt kapcsolatok háttérbe szorulnak (felejtés), bár soha sem tűnnek el.
- Engedjünk meg csoportképzést tetszőleges atomokra. Nevezzük őket komplex tudáselemnek vagy kontextusnak.
- Az egyszerű és a komplex tudáselemekre is ugyanazok a műveletek vonatkozzanak.
- Engedjük meg egymásnak ellentmondó atomok létét a bázisban.
- Ne tekintsük rossz működésnek, ha ugyanarra a kérdésre nem mindig ugyanazt a választ kapjuk.

### Tudásmennyiség, távolság

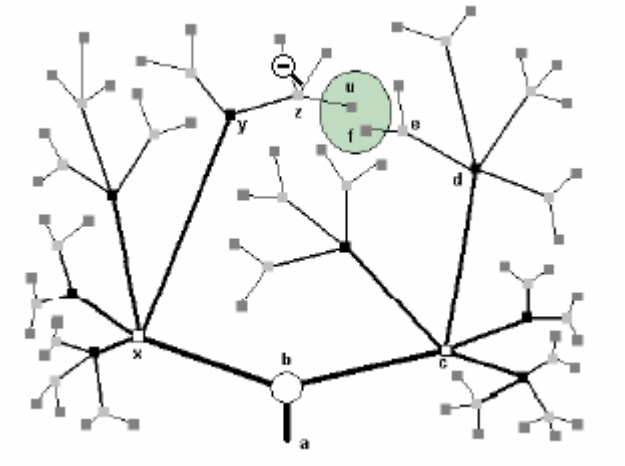
Tisztázásra vár, hogy az atomok és a kapcsolataik milyen szerkezetűek. Először vizsgáljuk meg a hierarchikus modellt. Jól ismert fa struktúra a tudományok hierarchikus felosztása (2. ábra) fizikára, kémiára, biológiára, geológiára, illetve ezeken belül a megfelelő szakterületekre (pl. fizika: mechanika, termodinamika, kvantummechanika, relativitás elmélet, vagy biológia: növénytan, állattan, genetika

vagy geológia: ásványtan, kőzettan, földtan).




2. ábra
A tudomány hierarchikus felosztása szaktudományokra

Alkossuk meg a tudás fáját atomokból és ezek kapcsolataiból. Az ágak a kapcsolatrendszert, a csomópontok az atomokat jelentik (3. ábra). Definiáljuk továbbá a kontextus fogalmát, amely tetszőleges atomokból álló csoport. Az alapelvekben kikötöttük, hogy ne különböztessük meg az egyszerű és a komplex elemeket, így a kontextusnak egyaránt lehet eleme egy egyszerű atom és egy másik kontextus, vagyis egy komplex tudáselem is.




3. ábra
A tudás fája

Jelölje $a_i^j$ az i-edik atomi ismeretet, a j-edik kontextusban, amiben N darab atom van. A kontextus tudásmennyisége legyen az atomok tudásának az összege:

$$k^j = \sum_{i=1}^{N} a_i^j$$

Jelölje $l_{ij}$ az $a_i$ és $a_j$ atomok közötti kapcsolat erősségét, amit jellemezzünk egy számmal, amely minden alkalommal, amikor végigmegyünk a két csúcspont közötti élen, inkrementálisan növekszik $l_{ij}= l_{ij} +1$, ha jó eredményre vezetett a vonalon való végighaladás, vagy $l_{ij}= l_{ij} -1$, ha rossz eredményre vezetett a vonalon való

végighaladás.

Vizsgáljuk meg, hogy milyen távolság definíciót használhatunk a 3. ábra u és f pontjaira. Az euklideszi távolság szerint ez két közeli pont, de mivel köztük nem áll fenn közvetlen kapcsolat, ezért jó távolság definíció az ágak mentén értelmezhető.

Legyen $a_i$ és $a_j$ a tudásbázis két atomja, amelyek között m=j-i darab atomon keresztül vezet az út. Legyen a két atom távolsága a köztük lévő kapcsolatok erőssége $l_{ij}$ összegének reciproka, vagyis minél erősebb a kapcsolat két csomópont között, annál közelebb vannak egymáshoz:

$$d_{ij} = 1 / \left( \frac{\sum_{k=i}^{j-1} l_{k(k+1)}}{m} \right)$$

Az evolúció során a reakcióidő fontos szerepet kapott, vagyis alapvető jelentőségű, hogy a bejárás a legrövidebb idő alatt történjék meg.

**Nem hierarchikus tudásbázis**

Vizsgáljuk meg az atomok tárolásának egy nem hierarchikus módját. Jelöljük **K**-val a tudásbázist, amely n számú atomból áll, $a_i$ és $a_j$ eleme **K**, és amelyet nevezzünk tudásmátrixnak.

$$\mathbf{K} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}$$

Az atomok kapcsolatban állnak más **K**-beli atomokkal. Jellemezzük $a_i$ és $a_j$ atomok kapcsolatát a $l_{ij}$-vel, ahol

$$l_{ij} = \begin{cases} 1 & \text{ha } i = j \\ 0 & \text{ha } i \neq j \text{ és nincs kapcsolat } a_i \text{ és } a_j \text{ között} \\ \neq 0 & \text{ha van: } u \text{ eredményes és } v \text{ eredménytelen bejárás után } l_{ij} = u - v \end{cases}$$

Rendezzük mátrix formába az atomok kapcsolatrendszerét, amit nevezzünk kapcsolat mátrixnak, és jelöljük **L**-lel. A mátrix elemei az atomi pontpárok kapcsolatát jellemző számok ($l_{ij}$=1). Ez a mátrix diagonális ($l_{ii}$=1), amely leírja az atomok kapcsolatrendszerét.

$$\mathbf{L} = \begin{pmatrix} 1 & l_{12} & \dots & l_{1n} \\ l_{21} & 1 & \dots & l_{2n} \\ \vdots & \vdots & \ddots & \\ l_{n1} & l_{n2} & \dots & 1 \end{pmatrix}$$

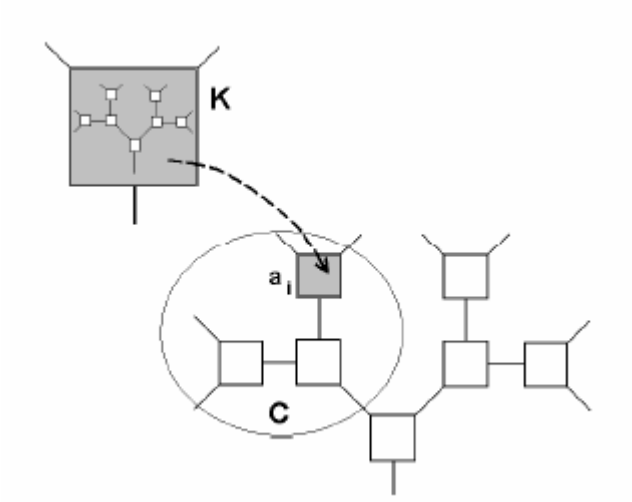
**A K és L mátrixok tulajdonságai és műveletei**

- Minden atomnak legalább egy kapcsolata van, vagyis nem létezhet kapcsolat nélküli atom.

- **K** bármely atomjából (részhalmazából) alkotható egy mátrix, amelyet nevezzünk kontextusnak.

- Egy atom több kontextusban is lehet atom.

- Egy kontextusnak egy másik kontextus is lehet atomja, például egy másik mátrix (kontextus) determinánsa, nyoma, elemeinek összege legyen a kontextus elemeiből levezetett tudásatom (4. ábra). Így minden atom egy fekete doboz, ami vagy tényleg atomi ismeret, vagy egy komplex tudáselem, ami mégis be tud épülni valamelyik kontextusba.




4. ábra

A kontextus, mint komplex tudásatom, tetszőleges összetettségű struktúra megalkotását teszi lehetővé, ami vélhetőleg skálafüggetlen tulajdonságokat is fog mutatni, ha elég nagyméretű lesz, és folyamatosan növekedni fog. A példában a **K**-val jelölt kontextus, mint komplex elem, a **C** jelű kontextusnak $a_i$ atomja.}

- A tudásbázis egésze is egy kontextus.

- Új atom beszúrása csak egy másik atomhoz történő kapcsolódás által tehető meg. Ez egyben azt is jelenti, hogy az atom a létrehozásának ,,körülményei'' is megjelennek a tudásbázisban, nemcsak a kapcsolódó másik atom, hanem a kontextus által is. Ennek a körülménynek kereséskor lehet jelentősége, mert a kontextus alapvető jelentőséget kaphat az asszociációs gondolkodásban.

- **K** és **L** mátrixok erősen függnek egymástól elvégre $l_{ij}$ mondja meg, hogy mely $a_{ij}$ atomi ismeretek kapcsolódnak egymáshoz az adott kontextuson belül. Akár egyetlen 3D-s mátrixban is tárolhatnánk az atomokat és kapcsolataikat. Bővítsük a mátrixot egy harmadik komponenssel, amely egy másik kontextusra mutat, ha az atom komplex elem. Így tehát minden atom legyen egy számhármas: 1. atomi tudás, 2. kapcsolódás más atomokhoz, 3. mutatás egy másik kontextusra. Előfordulhat ugyanis, hogy nem elegendő a kontextust reprezentáló elem, hanem ténylegesen be is kell lépni a kontextusba.

- Négyféle atom van:
    - tényleg oszthatatlan, elemi ismeret
    - olyan elemi ismeret, amely egy másik kontextusnak is tagja, vagyis egy másik kontextusra utal
    - egy reprezentáns tudáselem, amely egy másik kontextusnak a reprezentánsa (determinánsa, nyoma, stb), de nem tartozik hozzá bejárható kontextus.
    - Egy komplex tudáselem, amely maga is egy kontextus

## A tudásbázis fejlődése

- A tudás evolúciós úton, fokozatos fejlődés révén jön létre.

- Mivel a tudás egyéni evolúciós termék, ezért függ az egyén életútjától, vagyis a ,,neveltetéstől". Egyedről egyedre más és más lesz.

- A körülmények hirtelen változása a tudásbázis már eddig létrejött atomjait érintetlenül hagyja, de újabb atomok hozzáadásával azt bővíti, esetleg olyanokkal is, amelyek az eddigieknek ellentmondanak.

- Az idő nem megfordítható, vagyis a tudásbázis épülése egyirányú folyamat, törölni nem lehet belőle, persze felejtés által a jelentősége csökkenhet egy-egy atomnak vagy kontextusnak, akár a nullára is, ami majdnem olyan, mint a törlés, de azért mégsem az.

- Mivel a visszaigazolás kívülről jön, ezért a kérdés és a rá adott válasz is a tudásbázis elemévé kell váljon. Ezt egyelőre nem látom át, hogy hogyan.

- Következmény: ha bármilyen elemhalmazból alkotható kontextus (mátrix), akkor tetszőleges összetettségű rendszer alkotható.

- Engedjünk meg kétféle üzemmódot. Az egyik az aktív, amikor az érzékelések és az adatok tárolása történik, a másik a passzív, amikor nincs érzékelés, de a beérkezett adatok feldolgozása, rendszerezése, strukturálása folyik. A passzív üzemmód adhat lehetőséget arra, hogy a tapasztalatok átértékelése révén új kontextusok jöjjenek létre a meglévő atomokból. Ez teremthet lehetőséget az igazi okulásra, alapvető felismerésekre, vagyis a fejlődésben rendszeresen előforduló evolúciós ugrásokra.

- Intelligens lénytől kapott visszaigazolás gyorsan növeli a tudásbázis intelligenciáját, a környezettől kapott egyszerű tapasztalat lassan, elvégre az intelligens lény válasza is valamiféle beinkludálható kontextusnak tekinthető.

## Keresés a kontextusban

- Csak nagyon egyszerű algoritmus lehet a jó választ megtaláló eljárás (pl. legrövidebb idő elve, legerősebb kapcsolat preferálása), mivel az evolúció kezdetén csak meglehetősen egyszerű folyamatok zajlottak le a lények belsejében.

- Nincs beprogramozott algoritmus, amely megmondaná a jó válasz kiválasztásának módját. Megerősítés csak kívülről jöhet, vagyis a keresési algoritmus haladási irányát a tapasztalatból származó visszaigazolások által beégetett utak mutatják meg (pl. a legrövidebb idő alapján). Nincs tehát semmilyen optimalizációs algoritmus. A legtöbbet használt út a leggyorsabb (a sztereotip válaszok többnyire működnek, előítéletek), tehát ezeket kell preferálni.

- Az egyes utak jutalmazása külső megerősítés, tapasztalat útján történik (pl. a rossz válasz negatív következményekkel járhat, meg kell jegyezni a kérdést, és a választ is?)

## Egyelőre nem világos problémák

- Fontos lenne, hogy a tudásbázis időként át legyen értékelve, az elemekből új kontextusok jöjjenek létre pusztán az átértékelés által. Ezek át kell, hogy alakítsák a bejárást is. A tapasztalatok összegzése a belső struktúra átalakulásával kell járjon. Egyelőre nem látok okot rá, hogy ez miért következne be, márpedig e nélkül nehéz elképzelni az okosodás folyamatát. A pihenő, passzív üzemmód persze adhat módot erre a folyamatra, de vajon mi lesz a rendező művelet célja (idő szerinti optimalizálás)?

- Feltehetőleg nem hagyható figyelmen kívül, hogy sokaságban léteznek a lények. Kezdetben biztosan elegendő erőforrás állt rendelkezésre, mivel nem voltak sokan a kezdeti lények. Valószínű, hogy a létszám növekedésével az egyszer előállt verseny helyzet alapvetően átalakította a feltételeket. Nyilvánvalóan egész más tudásbázis fog kialakulni, ha korlátlanul van elég erőforrás, mint ha nincs, és versengeni kell értük. Pontosabban fogalmazva a kezdeti nem versengő tudásbázis kiegészül a versenyhez igazodó atomokkal és kontextusokkal.

- Mivel egyetlen atom sem törlődik, kérdés, hogy mi történik a csak elfelejtett atomokat tartalmazó kontextussal. Az előbbiből következik a kérdés, hogy létezhet atomok nélküli kontextus? A válasz nem, de elfelejtett atomokból álló igen.

## A tudásbázis létrehozása

A fentiekből világosan kiolvasható, hogy nem adható meg olyan algoritmus, amely alapján egy adott mennyiségű tudáselem (atom) intelligens rendszerbe szervezhető lenne. Az evolúciós fejlődés rekurzív jellege miatt nem mondható meg, hogy mikor melyik tudáselemnek kell következnie az építkezés során. Az is következik a fentiekből, hogy ugyanannyi tudáselemből az ,,összeszerelés'' módjától függő strukturális különbségek miatt eltérő képességű rendszer jöhet létre. Ez éppen azt jelenti, hogy a környezeti különbségek, az egyéni ,,életút'' eltérő tudású digitális lényeket eredményezhet.

### Szintetikus világok

A fent leírtak megvalósítása nem egyszerű. Egyelőre csak az a lehetőség látszik megvalósíthatónak, hogy szoftveresen alkossunk egy egyszerű, mesterséges világot, egyszerű környezettel. Alkossunk meg olyan szoftver objektumokat, amelyek ebben

a szintetikus világban próbálnak megélni, túlélni, és a környezet megfigyelése és kiértékelése által építeni a saját tudásbázisukat. Nevezzük el az objektumokat digitális evolúciós gépeknek (DEM: Digital Evolution Machine).

**Mesterséges környezet**

A következő egyszerű világot kellene felépíteni: legyen néhány környezeti paraméter, ami valamiféle szintetikus időjárás volna, mint például vízhőmérséklet, sótartalom, napfény, vízmélység determinisztikus és sztochasztikus módon változva. Erőforrások megszerzése a cél, amelyet a környezet változásai segíthetnek vagy akadályozhatnak. Vizsgáljuk meg, hogy sok objektumot létrehozva milyen tudásbázist fognak építeni a tapasztalataik alapján.
A mesterséges világ legyen nagy, ahol helytől függően is változnak a paraméterek az időbeli változásoktól függetlenül. Ez tehát azt jelenti, hogy egyidőben több, különböző környezet is jelen van a különböző helyeken.

**A digitális evolúciós gép**

A peremfelételek alapvetően befolyásolhatják a gépek működését. Biztosan másképpen működnek, ha korlátlanul állnak rendelkezésre erőforrások, mintha versenyezni kell értük. A korlátlan esetben elegendő, ha a tudásbázisuk az erőforrások megtalálását segíti, míg a versengő esetben nemcsak megtalálni kell az erőforrásokat, hanem a versenytársakat is le kell győzni valamilyen módon.

Egyszerre sok gép létezzen azért, hogy az eltérő környezetből származó eltérő tudásbázisú gépek jöjjenek létre. Ez a konstrukció biztosítja a sokféle irányú fejlődés lehetőségét. A fejlődési folyamat nyilvánvalóan nem lineáris, mivel az eltérő környezetben ,,nevelkedett'' gépek másképpen fognak reagálni a környezet megváltozásaira. Jelentősebb környezetváltozáshoz való alkalmazkodás az egyik tudásbázisnak kisebb a másiknak nagyobb kihívást jelenthet. Ez lesz majd az elhullás jelensége, ahol is a változásokhoz alkalmazkodni nem képes gépek ,,elhullanak'', és megmaradnak az ,,életképesek''.

**Az alapelvekből nem következő, de szükséges működési elemek**

- 1. verzió: legyen a gépeknek futamidejük, amely alatt gyarapíthatják a tudásbázisukat. Leállásuk után tudásbázisukat átadják az ,,utód'' gépeknek, akik arról a tudásszintről indulnak, ahol a szülők megálltak. Így óriási tudású

lények jöhetnének létre. Ez a verzió olyan gépeket eredményezne, mint ha örökké ,,élő'' gépek gyűjtenék a tapasztalatokat.

- 2. verzió: legyen a gépeknek futamidejük, amely alatt gyarapíthatják a tudásbázisukat. Leállásuk után tudásbázisuk valamilyen töredékét átadják az ,,utód'' gépeknek, akik erről a tudásszintről indulnak. (Gondoljunk csak arra, hogy mi haszna lenne ma egy páncélos lovag bajvívásban szerzett tudásának? Semmi, ezért tárolni sem érdemes távoli ősök tudását).

- Legyen továbbá minden gépnek ,,karbantartási'' ideje (alvás), amikor nem kell fogadniuk új adatokat, vagyis a receptorok nem működnek. Ilyenkor kell feldolgozni a ,,napi'' tudást, amely még kiértékeletlen, struktúrálatlan, és amely a tudásbázisba történő beépítés után válik igazán tudáselemmé, atommá.

- Engedjük meg továbbá, hogy az utód gép átdefiniálhassa a számára ideálisnak mondott paramétereket. A mértéke persze kérdéses, de kisebb mértékű változtatás kívánatos lenne, elvégre az evolúció apró lépésekben változtatja az élőlények paramétereit. Ezzel a lehetőséggel az lenne biztosítható, hogy az intelligencia fejlődésével a lények a megismerési tevékenységük által csatolnak vissza testi tulajdonságaik kialakítására. Ilyen folyamat megfigyelhető az élővilág evolúciója során.

Az előbbi néhány ponttal az a baj, hogy a dolgozat elején lefektetett alapelvekből nem következik egyik sem. A véges élettartam éppúgy nem, mint az aktív és passzív üzemmódok váltogatása. Márpedig éppen az a cél a DEM megalkotásával, hogy az egyszerű alapelvek alapján épülő tudásbázis elvezessen az intelligens lény kialakulásához.

## Konszolidáció

A valódi intelligencia kialakulásához mesterséges környezetben, nagy tömegű digitális evolúciós gép működtetésén keresztül vezet az út. A nem világos kérdések is csak akkor fognak érthetőbbé válni, ha már lesznek működési tapasztalatok. Látni kell azonban azt is, hogy a környezet megalkotása alapvetően fogja befolyásolni a DEM-ek működését. Túl egyszerű környezet feltehetően egyszerű DEM-eket fog eredményezni, vagyis a mesterséges környezet megalkotásában a valóságos környezet összetettségét kell elsősorban szimulálni.

# Irodalomjegyzék


1. William K. Pratt, Digital Image Processing, John Wiley & Sons, New York, 1991
2. M. Kuwahara, K. Hachimura, S. Eiho, and M. Kinoshita. Digital Processing of Biomedical Images.,Plenum Press, New York, NY, 1976
3. N. Otsu, A threshold selection method from gray-level histograms, IEEE Trans. Sys., Man., Cyber., vol. 9, 1979
4. P. Liao, T. Chen and P. Chung, A Fast Algorithm for Multilevel Thresholding, Journal of Information Science and engineering 17 2001.
5. Canny, J., A Computational Approach To Edge Detection, IEEE Trans. Pattern Analysis and Machine Intelligence, 8:679-714, 1986
6. Lindeberg, Tony "Edge detection and ridge detection with automatic scale selection", International Journal of Computer Vision, 30, 2, pp 117--154, 1998
7. Lam L., Lee S. W., Suen C. Y. (1992): Thinning methodologies - a comprehensive survey. IEEE Trans. on Pattern Analysis and Machine Intelligence, Vol. 14, No. 9, pp. 869-887
8. Lam L., Lee S. W., Suen C. Y. (1992): Thinning methodologies - a comprehensive survey. IEEE Trans. on Pattern Analysis and Machine Intelligence, Vol. 14, No. 9, pp. 869-887.
9. S. Kirkpatrick and C. D. Gelatt and M. P. Vecchi, Optimization by Simulated Annealing, Science, Vol 220, Number 4598, pages 671-680, 1983.
10. D. H. Douglas and T. K. Peucker. Algorithms for the reduction of the number of points required to represent a line or its caricature. The Canadian Cartographer, 10(2):112--122, 1973.
11. Graham, R.L. An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set. Information Processing Letters 1, 132-133 1972.
12. A.M. Andrew, Another Efficient Algorithm for Convex Hulls in Two Dimensions, Info. Proc. Letters 9, 216-219, 1979.
13. G. Toussaint, Solving geometric problems with the rotating calipers, Proc. MELECON '83, 1983
14. ESRI Shapefile Technical Description - ESRI White Paper, July 1998
15. Gonzalez, Woods Digital Image Processing, Prentice Hall, 2002
16. Smith: Digital Signal Processing, Elsevier Science, 2003
17. Allen, Mills: Signal Analysis, Wiley, 2004
18. Mix, Olejniczak: Elements of Wavelets for Engineers and Scientists, 2003
19. Newman, Barabasi, Watts: The Structure and Dynamics of Networks, Princeton University Press, 2006
20. Rigaux, Scholl, Voisard: Spatial Databases with Application to GIS, Morgan Kaufmann Publishers, 2002