Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Digitális képfájlokba rejtett adatok visszanyerhetőségének vizsgálata

TDK dolgozat

Készítette:
Simon Gergely

Konzulens:
Dr. Max Gyula

2011

Tartalomjegyzék

1. Bevezetés - A szteganográfia alapjai **2**

2. Adat rejtése tömörítetlen hordozóformátumokba **4**
2.1. Szöveg elrejtése 4
2.1.1. A mérések 5
2.2. Kép rejtése 7
2.2.1. Mérési eredmények 8
2.3. A megfelelő képek kiválasztása 9

3. A JPEG-tömörítés során fellépő nehézségek **12**
3.1. A JPEG-szabvány 12
3.1.1. A JPEG kódolás és dekódolás lépései 12
3.1.2. A fájlformátum 14
3.2. A tömörítés által okozott hibák 16
3.3. Kisebb arányú karakterhibát garantáló módszerek 18
3.3.1. A különböző hibajavító kódolások hatékonysága 19

4. Adat rejtése JPEG-tömörített hordozókba **22**
4.1. Az adatrejtés elve 22
4.2. A mérések 24

5. Az algoritmusok biztonsága, felhasználása **26**
5.1. Alkalmazási területek 26
5.2. A rejtett adatok biztonsága 26

A. Táblázatok **30**

B. Képek **32**

1. Bevezetés - A szteganográfia alapjai

Az információ elrejtése, titkosítása mindig fontos szerepet játszott az emberiség életében, hiszen a hadvezérek, politkusok nem szerették volna, hogy az állam számára fontos információk avatatlan kezekbe jussanak. Hasonlóan sokszor magánjellegű levelezések is folytak kódolt formában. Az egyik első említés ilyesfajta eszközök használatáról az ókori Rómából ered, és használója nyomán a Caesar-kód nevet viseli. Történetírók feljegyzéseiben megfigyelhetünk még más eljárásokat is, melyek ebben az időben kezdtek elterjedni. Ekkoriban választották szét az üzenet titkosítását (kriptográfia) és elrejtését (szteganográfia), ezt jól bizonyítja mindkét szó görög eredete. A dolgozatomban a szteganográfia modern, 21. századi eszközök által biztosított lehetőségeit vizsgálom, ezekhez többnyire előzményt biztosítanak a már történelminek mondható eljárások[1]:

- Viasszal bevont fatábla
  Az ókori Spártában a perzsák támadásáról egy fatáblára vésett szövegben adtak tájékoztatást. A szöveg elrejtése okán a táblát viasszal vonták be, így annak megolvasztása után vált csak újra olvashatóvá az üzenet

- Tetovált fej
  Hasonlóan az előző esethez, csak itt az információt a futár leborotvált fejére tetoválták, majd megvárták, hogy a haja visszanőjön, és így bocsátották útjára. Nyilvánvaló hátrány, hogy hosszú időbe telik az üzenet „megírása”, illetve, hogy bárki egyszerűen felfedezheti azt

- Láthatatlan tinta
  Talán közismert, amikor egy ártalmatlan szövegre, levélre olyan anyaggal írnak, mely megszáradás után színtelen, azonban hő hatására elszíneződik. Ilyen például a citromlé vagy tej, de a CIA nemrég nyilvánosságra hozott első világháborús dokumentuma[11] is tartalmaz ilyen vegyületeket. Modernebb változatában UV-festékkel történik az üzenet írása, és az olvashatóságot UV-fény biztosítja

- Mikropont
  Ebben az esetben az információ egy szövegbeli pont méretére van lezsugorítva, így nem szembetűnő, de nagyító segítségével könnyedén elolvasható. Az eljárást a II. világháború során használták

A számítástechnika, és digitális eszközök elterjedésével más lehetőségek is adódtak a fájlok megfelelő módosítására, ezek előnye egyrészt, hogy könnyen változtatható algoritmussal rendelkeznek, másrészt, hogy elegánsabbnak mondhatók, tehát nincs szükség semmiféle vegyületekre, tetoválásra, egyéb fizikai eszközre. Néhány ismert eljárás[1]:

- Whitespace/Formátum kódolás
  Formázatlan szövegfájlok esetén a whitespace karakterek (legtöbbször szóközök) számának változtatása nem szembetűnő, azonban ezek algoritmikusan gyorsan és hatékonyan módosíthatók. Legegyszerűbb esetben egy előfordulási helyen egy szóköz

beszúrása (vagy változatlanul hagyása) már 1 bit rejtését biztosítja. Formázott szövegek esetén szintén nem feltűnő az egyes karakterek betűméretének csekély megváltoztatása (legtöbbször az írásjeleket, vesszőt, mondtatvégi pontot), amely egy előfordulási helyen szintén néhány bit információ hordozására ad lehetőséget.

- Utasításkészlet-kódolás
  Minden processzor utasításkészletében megtalálható az üres utasítás, mely effektíven semmiféle műveletet nem végez, valamint számos esetben olyan utasítások is találhatók, melyek ugyanazt a funkciót biztosítják, más néven. Ezek felhasználásával akár olyan futtatható állományokat is készíthetünk – vagy úgy módosíthatunk egy már meglévőt –, hogy annak funkciója teljes mértékben változatlan marad, mégis tud plusz elrejtett adatot hordozni.

- Leírónyelv-manipuláció
  Az előző két eset keverékeként tekinthetünk a leírónyelv-manipulációra, mely az ún. tag-eket (HTML esetén a < > jelek közé zárt parancsokat) módosítja. HTML esetén ezek nem érzékenyek a kis- illetve nagybetűk közötti különbségekre, így pl. egy <html> vagy <body> tag 4-4 bit elrejtésének lehetőségét kínálja fel. Vigyáznunk kell azonban arra, hogy nem minden tag írható át, így pl. JavaScript használata esetén.

- Redundáns bitek módszere
  A digitális adathordozók (képek, hangfájlok, videók stb.) többnyire nagyobb bitmélységgel kerülnek tárolásra, mint az szükséges lenne. Így a redundáns biteket (mely tömörítetlen fájlok esetén a legalsó néhány bit, tömörítettebbek esetén pedig az algoritmustól függő bitek) megkeresve, azokat szintén felhasználhatjuk ilyen adatrejtési célokra.

Dolgozatomban a felkínált széles skáláról az utolsóként említett redundáns bitek módszerét fogom vizsgálni, tömörítetlen, valamint tömörített adatformátumokra, illetve azt, hogy tömörítés során a rejtett adat milyen mértékben torzul. Továbbá megvizsgálom, hogy mitől függ az elrejtett adat mennyisége, hogyan lehet algoritmikus úton az alkalmas képfájlokat megkeresni, milyen módszerekkel lehet továbbá bennük a legnagyobb adatrejtési lehetőséggel bíró területet megtalálni. Végezetül a felhasználási területekről, és a módszer biztonságáról, sebezhetőségéről fogok említést tenni.

Az eljárások teszteléséhez saját, C# nyelven írt programokat használtam, melyek többnyire a beépített könyvtári függvényeket használják a képek tárolásához, mentéséhez. Egyedüli kivételt jelent ez alól a JPEG-kódoló/dekódoló, mely hiányában egy saját függvénykönyvtárral valósítottam meg az ezekhez kapcsolódó tesztelést. A programok futtatható formában az interneten is megtalálhatók a függelékben megadott linken.

2. Adat rejtése tömörítetlen hordozóformátumokba

Azokban az esetekben, mikor a multimédiás fájlok - legyen szó képről vagy hangról - (veszteséges) tömörítés nélkül kerülnek tárolásra, lesznek olyan redundáns bitek, amelyek nélkül (vagy megváltoztatásával) az adott fájl láthatóan (vagy hallhatóan) nem változik meg. Célunk ezeknek a biteknek a megkeresése, és olyan eljárások kidolgozása, melyek effektíven használják ki ezeket.

A további vizsgálataimat a manapság dominanciával rendelkező 24-bites RGB fájlokon végeztem, így ahol ezt külön nem jelzem, ezeket értem képfájl alatt. Ezen fájlokban három színkomponensből áll össze egy pixel színe minden esetben, ezek az értékek egyenként 8 biten vannak letárolva. Mivel így összesen $2^{3 \cdot 8} = 2^{24} \approx 16.8$ millió szín lehetséges, viszont az emberi szem mindösszesen kb. 8-10 millió színt tud megkülönböztetni,[10] ami jó közelítéssel a fele, így máris lehetőségünk nyílik egy bit elhagyására, azaz rejtett adat tárolására. Azonban ennél is kedvezőbb, hogy nem csupán a legalsó bit (LSB), hanem még további bitek módosítására van lehetőségünk, amennyiben a felhasznált kép emberi szemmel nézve megfelelő (részletesen a 2.3 fejezetben).

Így tehát azt fogom vizsgálni, hogy a tömörítetlen képek legalsó néhány bitjére milyen módon mekkora adatmennyiséget lehet elrejteni, ez mennyire észrevehető, illetve, hogy milyen módon lehet az alkalmas képfájlokat megkeresni, elemezni.

2.1. Szöveg elrejtése

A fent ismertetett módszert a következőképpen valósítottam meg: a képen egy kijelölt téglalap alakú tartományban található pixelekbe – rendre a bal felsőtől kezdve soronként haladva, egy pixelen belül R, G, B sorrendben – kerül a megadott szöveg elrejtése. Mivel kiolvasáskor egyrészt tudnunk kell, hol kezdődik a számunkra releváns adat, mekkora a befoglaló téglalap mérete (szélessége a fontos), illetve, hogy mekkora mennyiségű információt kell kiolvasni, ezért a beíráskor a szöveget egy headerben helyeztem el, melynek legfőbb részei az 1 táblázatban láthatók. Így a header kezdetét a 0xFF5346FF bájtsorozat jelzi, majd rendre a fentebb említett adatok követik, végezetül maga a szöveg. A 2 bájton tárolt szélesség maximális $256^2 - 1 = 65535$ pixel hosszú tartományt tesz lehetővé, míg a 3 bájton tárolt szöveghossz $256^3 - 1 \approx 16.8$ millió karakter elrejtését. Mindkettő elégséges tartományt biztosít, és minimális nagyságú (1 illetve 2 bájt még kevés lehetne).

SoF marker bájtok				Szélesség	A szöveg hossza (n)	Adat
0xFF	0x53	0x46	0xFF	*Big-endian*	*Big-endian*	
4 bájt				2 bájt	3 bájt	n bájt

1. táblázat. Az elrejtett adat formátuma

Mivel a szövegben magyar ékezetes karakterek támogatását is lehetővé kell tenni, illetve

ezeknek az Unicode értéke 2 bájt lenne, célszerűnek mutatkozott, hogy egy saját karaktertáblával dolgozzon a program, mely így az összes kis-nagybetűt, számot, és fontosabb írásjelet támogatja, és mindegyik mindössze 1 bájt helyet foglal. Az adatok elrejtését úgy vizsgáltam, hogy minden egyes színkomponens alsó k bitjét (ez azonos a három csatornára) módosítottam. Ennek alapján egy pixel $3k$ bitet tud hordozni, azaz egy mxn-es terület összesen $3knm$-et (nem számolva a header bájtokkal). Ez bájtok (b), azaz karakterek számában kifejezve (immáron a header járulékos karaktereit levonva) az (1) egyenlettel lehet.

$$b = \left\lfloor \frac{3knm}{8} \right\rfloor - 9 \tag{1}$$

Továbbá a szöveg i-edik bitjének helye is megadható a hordozó képen belül (pixel, színkomponens, bit pontossággal), így a gyakorlatban mindössze egy ciklus segítségével lehet az adatokat elrejteni. Feltéve, hogy a befoglaló téglalap offsetjét (eltolását a kép bal felső sarkához képest) o, a szélességét w, valamint a rejtési bitmélységet k jelölésekkel illetve a pixel helyét a (2) összefüggés adja meg, hiszen X irányban akkor lépünk a következő pixelre, ha az előzőbe már több bitet nem írhatunk, azaz az összes $3k$ lehetőségünket kihasználtuk, valamint vigyáznunk kell, ha ez a tartományon kívülre esik, ezért kell maradékot vennünk w-vel. Y irányban szimplán a teljes előző sor kihasználtsága esetén lépünk, ami $3kw$ bitenként fordul elő. A módosítandó színkomponenst a (3) összefüggés értéke határozza meg, mégpedig 0, 1 illetve 2 esetén rendre R, G és B, hiszen itt minden k bitenként van komponensváltás. A módosítandó bit pedig egészen egyszerűen a (4) képlettel számolható, amennyiben a biteket LSB-től, nullával kezdjük számozni.

$$\text{pixel}\left[o.X + \left\lfloor \frac{i}{3k} \right\rfloor \bmod w; \quad o.Y + \left\lfloor \frac{i}{3kw} \right\rfloor\right] \tag{2}$$

$$\left\lfloor \frac{i}{k} \right\rfloor \bmod 3 \tag{3}$$

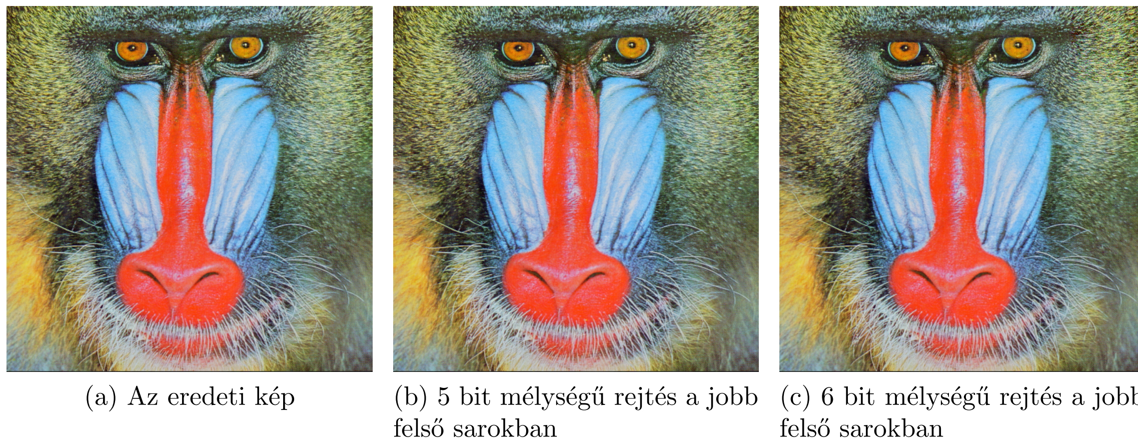
$$k - i \bmod k - 1 \tag{4}$$

2.1.1. A mérések

Méréseimhez a széles körben elterjedt két tesztképet, Lenát és a mandrillt használtam. Előnyük, hogy az előbbi egy portré lévén sok homogén (kisfrekvenciás) területet tartalmaz, míg az utóbbi szinte csak gyorsan változó, nagyfrekvenciás részekből áll. Így várható, hogy valamilyen különbséget lehet majd felfedezni az elrejtési mélységek tekintetében. (A képek megtalálhatók eredeti méretben a függelékben is, 15-20. ábrák)

A 2. képsorozat alapján tapasztalható, hogy a homogénebb területeken már kisebb mértékű rejtett adat is meglátható, míg a 1. képsorozat egészen nagy mennyiségű adatrejtés lehetőségét mutatja. Számszerűen kifejezve, a portréban a kb. 110×80-as téglalapra 3

bit módosításával $b \approx 9900$ karaktert, míg a mandrillra 5 bit módosításával a 100×140-es területre $b \approx 26250$ karaktert lehet beírni. Mivel ez utóbbi több, mint Rejtő Jenő: Tizennégykarátos autójának teljes első fejezete (~23000 karakter), így az algoritmus tesztelése céljából ezt rejtettem el a jobb felső sarokba. Érdekes eredményt adhat ebben az



(a) Az eredeti kép (b) 5 bit mélységű rejtés a jobb felső sarokban (c) 6 bit mélységű rejtés a jobb felső sarokban

1. ábra. A tömörítetlen hordozóba rejtéskor a nagyfrekvenciás kép több bit módosítást visel el

(a) Az eredeti kép (b) 3 bit mélységű rejtés a vállon, még nem látható (c) 4 bit mélységű rejtés a vállon, amely már észrevehető

2. ábra. Hasonló eljáráskor a homogénebb kép kevesebb bit módosítást visel el

esetben szürkeárnyalatos kép használata. Mivel ebben az esetben a három színkomponens megegyezik ($pixel[i,j].R = pixel[i,j].G = pixel[i,j].B$), és nagy valószínűséggel a rejtett adat bitjeire nem lesz igaz, hogy a három színkomponenst azonos módon változtassák, bekövetkezik az adott területen egy „elszíneződés”, azaz a szürke árnyalatai helyett színes pixelek is esetenként előbukkannak. Ez látható a függelékben, a 11. ábrán, azonban a 4 bites rejtés ellenére a változás még csekély a várttal szemben.

Az eredmények alapján általánosan elmondható, hogy az ilyen fajta adatrejtés magas

tömörítési mélységet kínál (még a legrosszabb esetben is 2-3 bitet, azaz 25-37.5%-ot, de akár 62.5%-ot), azonban a fájl mérete is nyilvánvalóan nagy lesz (néhány MB), illetve olyan formátumokat használ (pl. BMP), amelyek manapság már ritkán használatosak.

2.2. Kép rejtése

A szöveg elrejtéséből levont következtetés, miszerint még szélsőséges esetben is van legalább 2-3 szabadon módosítható bit, lehetőséget ad arra, hogy az egész képet felhasználva, egy kép a képben módszert is teszteljünk. Ennek első lépéseként, az adatrejtési bitmélység ismeretében (k, ami meghatározza, hogy hány alsó bit kerül módosításra) a beágyazandó kép színmélységét szükséges k-ra csökkenteni.
Ezt egy lineáris mintavételezéssel

$$\mathrm{RGB}_{\mathrm{uj}} = \mathrm{RGB}_{\mathrm{regi}} \frac{2^k}{256} \tag{5}$$

teszem meg. Majd a hordozókép adott pixelének alsó k bitjének eltávolítása után egyszerűen ezt az RGB_{uj} értéket hozzáadom. Ezáltal a kimeneti fájlban az alsó k bit valóban az elrejtendő kép csökkentett színmélységű változatát tartalmazza.
Továbbá a képeket úgy méretezem át, hogy a lehető legjobban kitöltsék a rendelkezésre álló helyet. Azaz a hordozó és elrejtendő kép (a továbbiakban $\mathrm{img}_{\mathrm{ho}}$ és $\mathrm{img}_{\mathrm{elr}}$) arányait ((6) egyenletek) figyelembevéve vagy a magasságát vagy pedig a szélességét használom ki teljesen a hordozóképnek. Feltéve, hogy $\mathrm{ratio}_{\mathrm{hordozo}} > \mathrm{ratio}_{\mathrm{elrejt}}$ – azaz arányaiban a hordozókép szélesebb, mint a beágyazandó kép – az elrejtendő kép pozíciói $(\mathrm{i}_{\mathrm{elrejt}}; \mathrm{j}_{\mathrm{elrejt}})$ megadhatók, a hordozó $(\mathrm{i}_{\mathrm{hord}}; \mathrm{j}_{\mathrm{hord}})$ koordinátáinak függvényében, (7) egyenletek alapján.

$$\begin{aligned} \mathrm{ratio}_{\mathrm{hordozo}} &= \frac{\mathrm{img}_{\mathrm{ho}}.\mathrm{Width}}{\mathrm{img}_{\mathrm{ho}}.\mathrm{Height}} \\ \mathrm{ratio}_{\mathrm{elrejt}} &= \frac{\mathrm{img}_{\mathrm{elr}}.\mathrm{Width}}{\mathrm{img}_{\mathrm{elr}}.\mathrm{Height}} \end{aligned} \tag{6}$$

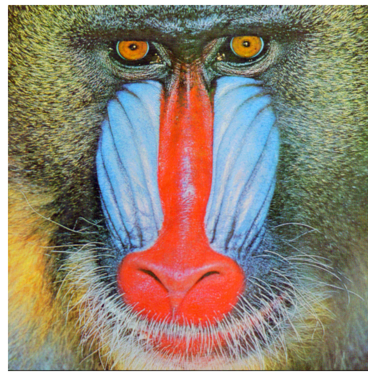
$$\begin{aligned} \mathrm{i}_{\mathrm{elrejt}} &= \mathrm{i}_{\mathrm{hord}} \frac{\mathrm{img}_{\mathrm{elr}}.\mathrm{Height}}{\mathrm{img}_{\mathrm{ho}}.\mathrm{Height}} + \frac{\mathrm{img}_{\mathrm{elr}}.\mathrm{Width}}{2} - \frac{\mathrm{img}_{\mathrm{elr}}.\mathrm{Height} \cdot \mathrm{img}_{\mathrm{ho}}.\mathrm{Width}}{2 \cdot \mathrm{img}_{\mathrm{ho}}.\mathrm{Height}} \\ \mathrm{j}_{\mathrm{elrejt}} &= \mathrm{j}_{\mathrm{hord}} \frac{\mathrm{img}_{\mathrm{elr}}.\mathrm{Height}}{\mathrm{img}_{\mathrm{ho}}.\mathrm{Height}} \end{aligned} \tag{7}$$

Hasonlóan a fordított $(\mathrm{ratio}_{\mathrm{hordozo}} < \mathrm{ratio}_{\mathrm{elrejt}})$ arányra is felírhatók ezek az összefüggések, természetesen akkor a szélesség lesz rögzített, a magasság pedig a dinamikusabban változó.
A módszer előnye ismét, hogy nem szükséges függvényekkel, illetve ciklusokkal meghatározni a következő elem helyét, hanem egyszerű összefüggésekkel explicit módon megadhatók.

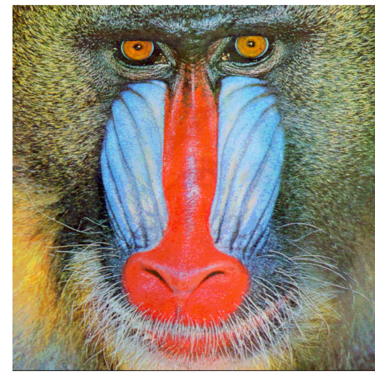
2.2.1. Mérési eredmények

Ehhez először az előbb említett két képet használom fel, a különböző bitmélységek hatása jól látható. Már 3 bit esetén is a kontrasztos elemek (pl. a szem) átütnek a hordozó képen (3. ábrasorozat *a* és *b* kép középső területein) . Ennek kivédésére kerülni kell az olyan hordozókat, melyeken összefüggő homogén területek (jelen esetben pl. a mandrill orra) vannak, ezek ugyanis nem fogják kellően fedni a rejtett képet. Ekkor még maga a rejtett kép sem elég részletes (3. sorozat *d*, *e* képek), a bitmélység további növelése ennek minőségét már számottevően nem javítja, viszont a hordozón látható lesz (3/c,f kép, a függelékben 21 és 22. ábra). Megfelelő hordozóval, és 4 bites kódolással elérhető, hogy nem látszik meg a módosítás, mégis a rejtett kép már elég részletesnek mondható, visszaolvasás után. Erre példa az erdős táj, mely szintén a már jól ismert portrét tartalmazza (4. ábra, a függelékben 23 és 24. ábrák).

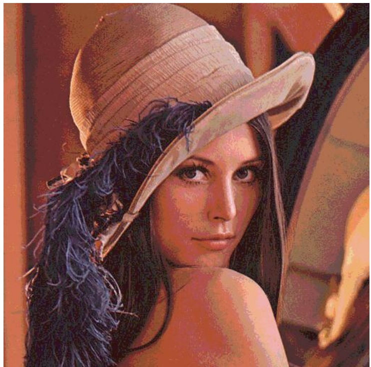
(a) A hordozó kép a rejtett adattal, 3 bit mélység



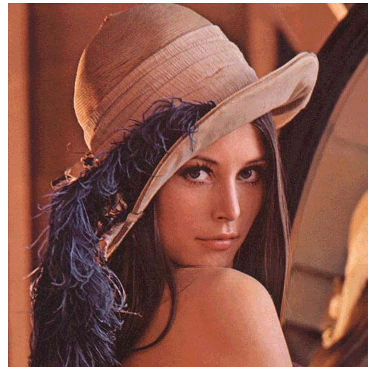
(b) A hordozó kép a rejtett adattal, 4 bit mélység



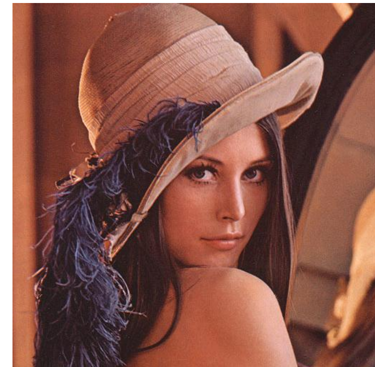
(c) A hordozó kép a rejtett adattal, 5 bit mélység



(d) A 3 bitre rejtett kép, visszaolvasás után



(e) A 4 bitre rejtett kép, visszaolvasás után, a minőség elfogadható



(f) Az 5 bitre rejtett kép, visszaolvasás után, a minősége nem sokkal jobb, mint a 4 bites változat

3. ábra. Kép a képben módszer esetén a különböző bitmélységek hatása

2.3. A megfelelő képek kiválasztása

Mivel célunkat képezi, hogy egy adott képbe a lehető legnagyobb adatmennyiség elrejtését érjük el, illetőleg hatékonyan (programkód segítségével) választhassuk ki az alkalmas képfájlokat, egy algoritmust dolgoztam ki, mely az előzetes megfigyeléseimen alapulva elvégzi a kívánt műveletet.

(a) Az eredeti erdős tájkép (b) A már módosított fájl (c) Az elrejtett kép minősége is elfogadható 4 bites kódolásnál

4. ábra. Kép a képben módszer hatékony alkalmazása

A képekbe történő adatrejtések során a manipuláció legtöbbször észrevehető az olyan helyeken, ahol nagyobb egybefüggő területek vannak, melyek egyszínűek, vagy esetleg egy kis gradienssel rendelkeznek, illetve ha szabályos mintázatúak. Éppen ezért előnyünkre válnak az olyan képek (képrészletek), melyek ilyen szempontból véletlenszerűnek mondhatók, azaz egy adott pixel a szomszédos pixelektől lehető legjobban eltér. Így az algoritmusom is ezeket az eltéréseket számolja, a következő módon: a kép minden egyes 3×3-mas blokkjára számolja a középső pixel összes többitől vett eltérését színkomponensenként (azaz a $|pixel[i,j].R - pixel[i,j-1].R|$ stb. különbségeket), majd ezeket a 2. táblázat alapján súlyozza, végezetül a színkomponensek közül így nyert legkisebb értékeket felhasználva egy színezett „térképet” biztosít a felhasználónak (az 5. ábrán láthatóak ilyen térképek).

(a) A már ismert madrill (b) És a neki megfelelő térkép (c) Valamint az erdő (d) És a térképe

5. ábra. Két kép, és a nekik megfelelő térképek

1	2	1
2	x	2
1	2	1

2. táblázat. A térkép készítéséhez használt súlyozások

Például a vörös színkomponensre kifejtve ez az érték az $(i;j)$ pixelre (feltételezve, hogy ez a pixel nem a kép széleire esik) a (8) egyenlettel írható le.

$$\begin{aligned} error_R = & \left|\frac{pix[i,j].R - pix[i+1,j].R}{6}\right| + \left|\frac{pix[i,j].R - pix[i-1,j].R}{6}\right| + \\ & \left|\frac{pix[i,j].R - pix[i,j-1].R}{6}\right| + \left|\frac{pix[i,j].R - pix[i,j+1].R}{6}\right| + \\ & \left|\frac{pix[i,j].R - pix[i+1,j+1].R}{12}\right| + \left|\frac{pix[i,j].R - pix[i+1,j-1].R}{12}\right| + \\ & \left|\frac{pix[i,j].R - pix[i-1,j-1].R}{12}\right| + \left|\frac{pix[i,j].R - pix[i-1,j+1].R}{12}\right| \end{aligned} \quad (8)$$

A térképen a legkisebb eltérések (amik számunkra a legkedvezőtlenebbek) piros, a közepesek sárga, valamint a legjobb pixelek zöld színt kaptak (0xFF0000–0xFFFF00–0x00FF00). Így könnyedén, ránézésre meghatározható, hogy melyik terület mennyire alkalmas céljainkra. Mivel a jelzett színek 511 egység nagyságú tartományt járnak be, azonban a különbségi értékek csak maximálisan 255 nagyságúak lehetnek, ezért egy exponenciális jellegű függvénnyel kötöttem össze a két tartományt. Ennek elsődleges célja az értékek megfelelő színekhez rendelése, illetve az eltérések markáns jelölése volt (kis eltérés már sárga vagy zöld színt eredményez, ami piros marad, az valójában nagyon rossz). A használt (9)-es függvény által visszaadott értékek által körülbelül jelzett lehetséges kódolási bitmélységek: piros 2-3 bit, sárga 3-4 bit, zöld 5, esetleg több bit. Jól látszik a térképek alapján, hogy míg az erdő teljes mértékben 4-5 bites mélységet tesz lehetővé, a mandrill esetében az orra kritikus helynek számít, mindössze 2-3 lehetséges bittel, és ezek jól egyeznek a mérések eredményével is, azaz, hogy a mandrill esetében valóban történik egy átlátszódás a középső területeken, az erdőnél viszont sehol sem.

$$\mathrm{color} = 511\left(1 - e^{-0.04\mathrm{error_i}}\right) \quad (9)$$

Ezen kívül a programok funkcióját képezi, hogy a térkép színezésén kívül az egyik legnagyobb (mivel az algoritmus heurisztikus elemeket is tartalmaz, a végeredmény nem feltétlenül a legjobb eredményt adja) zöld területet is megjelöli. Ehhez a következő algoritmust használtam:

- A térkép felosztása k részre, mind horizontálisan, mind vertikálisan
- A tartományokban „legzöldebb” (azaz legnagyobb eltérési értékű) pixelek megjegyzése ún. kiindulási magként (seed)

- A seedekből kiindulva megpróbáljuk a területet (ami ekkor még csak 1 pixel) kiterjeszteni. Amelyik irányban ez lehetséges, bizonyos paraméterek mellett (egyik a határ, ami alatti color értékek száma korlátozott (= threshold), a másik pedig ezeknek a color értékeknek (9) a maximális száma (= $\max_{\text{fail}}$)), azt az újabb területet egy láncolt listában tároljuk tartományonként

- A listában szereplő legrosszabb területek törlése, a már vizsgáltakban flag-gel jelezni ezt, a gyorsítás miatt

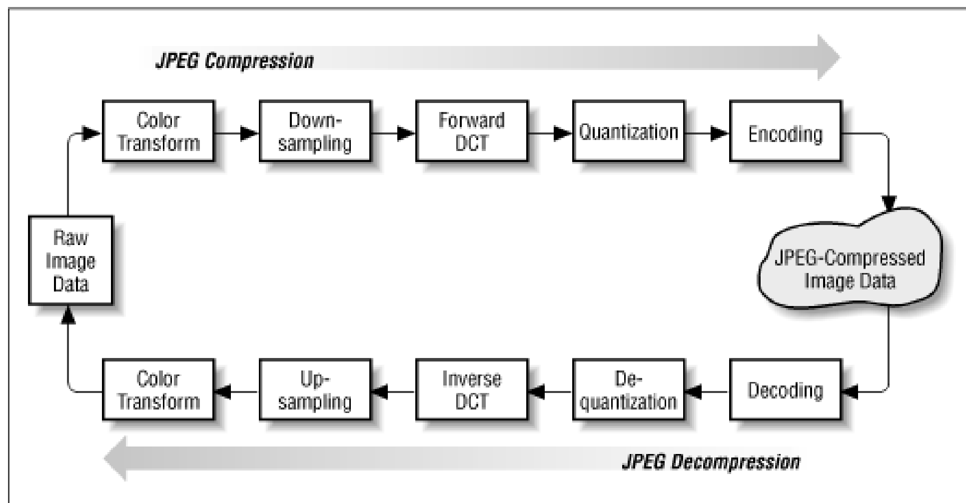
- Amikor már nincs ellenőrizetlen terület (azaz nem lehet tovább szélesíteni), a maximumot ezek közül kiválasztjuk és bejelöljük

A paraméterek véglegesítése során ügyeltem arra, hogy a futási idő és a jó közelítés is szem előtt maradjon. Így a végső tesztekben használt változókkal szinte minden esetben a legjobb terület került megjelölésre, és a kép méretétől függően ez csupán 3–5 mp-ig tartott. A változók: $k = 10$, azaz irányonként 10–10, így összesen 100 tartomány vizsgálata; threshold = 270, $\max_{\text{fail}} = 0.75$, azaz a 270 érték alatti color értékekből maximum 75% lehet egy területen, különben nem kerül felvételre a listába.

3. A JPEG-tömörítés során fellépő nehézségek

A tömörítetlen hordozókon végzett adatrejtés tulajdonságainak ismeretében célszerű valamilyen módon a fájl tömörítését megvizsgálni, hiszen ekkor jóval kisebb fájlméretet lehet elérni. Ehhez kézenfekvő a JPEG-tömörítést használni, és elemezni azt, hogy az elrejtett adat hogyan sérül (veszteséges tömörítés miatt valószínűleg sérülni fog), illetve milyen módon lehet azt esetleg visszaállítani. Mivel a következőkben a JPEG-tömörítés hatását fogom vizsgálni a BMP (vagy egyéb tömörítetlen) képformátumokba rejtett adatokra, elengedhetetlen a JPEG-kódolás elméletének (illetve az algoritmusok megvalósítása szempontjából a fájlformátumnak) az alapos ismerete. Így a következő fejezetben a kódolás/dekódolás lépéseit (6. ábra[9]), a másodikban a fájlformátumot mutatom be.

3.1. A JPEG-szabvány



6. ábra. A JPEG-kódolás és dekódolás lépései

3.1.1. A JPEG kódolás és dekódolás lépései

A JPEG-tömörítés során a következő lépések történnek meg:

1. Színtér konverzió (Colorspace conversion),
   melynek során az általánosan használt RGB térből a JPEG szempontjából előnyösebb YC_bC_r térre konvertáljuk a pixeleket. Itt a Y (luminance) fényerőnek felel meg, azaz gyakorlatilag egy szürkeárnyalatos képet ad. A kék-különbségi és piros-különbségi színkomponensek (C_b – Chroma blue difference, C_r – Chroma red difference) a pixel színárnyalatát határozzák meg. Az emberi szem számára minden esetben a körvonalak, így tehát a szürkeárnyalatos információk a legfontosabbak. Ez a színtér lehetőséget ad majd az Y komponens külön kezelésére, és a C_r, C_b értékek közül

néhány elhagyására. A (10,11) konverziós egyenletek[8] teremtenek kapcsolatot az RGB és YC_bC_r színtér között.

$$\begin{aligned} Y &= 0.299R + 0.587G + 0.114B \\ C_b &= -0.1687R - 0.3313G + 0.5B + 128 \\ C_r &= 0.5R - 0.4187G - 0.0813B + 128 \end{aligned} \tag{10}$$

$$\begin{aligned} R &= Y + 1.402(C_r - 128) \\ G &= Y - 0.34414(C_b - 128) - 0.71414(C_r - 128) \\ B &= Y + 1.772(C_b - 128) \end{aligned} \tag{11}$$

2. Színblokk átlagolás (Chroma subsampling),
   a következő pontban látni fogjuk, hogy a transzformáció egysége nem a pixel, hanem 8x8 pixel alkotta blokk. A színblokk átlagolás során a felhasználó megadhatja, hogy vízszintesen illetve függőlegesen hány blokk kerüljön összevonásra. Ez az Y komponenseket nem érinti, a színkomponensek pedig átlagolásra kerülnek az érintett blokkokban (természetesen csak az azonos pozícióban levők). A legelterjedtebb mintavételezések:[?]

   - 4:4:4
     Nem történik mintavételezés, minden blokk érintetlenül marad
   - 4:2:2
     Vízszintesen 2 blokk kerül összevonásra (2-2 érték átlagolódik), függőlegesen nincs mintavételezés
   - 4:2:0
     Mind függőlegesen, mind vízszintesen 2-2 blokk összevonásra kerül. Így 4-4 érték átlagolódik

   Alapesetben ezek közül is a legtöbb képkezelő a 4:2:0-ás mintavételezést használja, illetve külön opcióként választható a mintavételezés kikapcsolása (Disable Chroma Subsampling), ez nyilvánvalóan a 4:4:4-es esetnek felel meg.

3. DCT transzformáció (Discrete Cosine Transformation),
   a (12) konverziós egyenletekkel valósítható meg egy $N \times N$-es blokkon kétdimenziós esetben,[6] ahol $\alpha_i = \sqrt{\frac{1}{N}}$, ha $i = 0$ és $\alpha_i = \sqrt{\frac{2}{N}}$, amennyiben $i \neq 0$, valamint $Y(m,n)$ a transzformáció utáni amplitúdót, $X(k,l)$ az eredeti amplitúdót jelöli.

$$\begin{aligned} Y(m,n) &= \alpha_m \alpha_n \sum_{k=0}^{N-1} \sum_{l=0}^{N-1} X(k,l) cos\left(\frac{2k+1}{2N} m\pi\right) cos\left(\frac{2l+1}{2N} n\pi\right) \\ X(k,l) &= \sum_{m=0}^{N-1} \sum_{n=0}^{N-1} \alpha_m \alpha_n Y(m,n) cos\left(\frac{2k+1}{2N} m\pi\right) cos\left(\frac{2l+1}{2N} n\pi\right) \end{aligned} \tag{12}$$

Az előző pontokban ismertetett konverziós lépések már kínálnak egy kis mértékű tömörítést. Azonban a JPEG magját képező DCT transzformáció, mely során a DFT-hez hasonlóan egy másik tartományba konvertáljuk az adathalmazunkat, további előnyökkel szolgál, melyek közül a legfontosabbak:

- Az algoritmus nem komplex számokkal dolgozik, így egyszerűbb, gyorsabb a műveletvégzés — adott N-re (jelen esetben $N = 8$) a szummákban szereplő együtthatók előre kiszámolhatók, így már csak szorzások és összeadások maradnak fenn
- Mivel valós számtesten végezzük a transzformációs műveleteket, ennek következtében az eredmény is valós, így feleannyi tárolási hely kell, mint a DFT esetén
- A konverziós térben a nagy amplitúdójú elemek az origó közelében helyezkednek el, az energia nagy része ide koncentrálódik, ami szintén lehetőséget ad számtalan elem elhagyására (erről bővebben a következő pontokban)

4. Kvantálás (Quantisation)
   A DCT során keletkező blokkok a DCT sajátosságai miatt a $(0, 0)$ elem körül (mely a DC értéke a blokknak) tartalmazzák a legnagyobb értékeket, az $(N-1, N-1)$-es (legnagyobb frekvenciájú) tag felé haladva erőteljesen csökken az elemek nagysága, illetve az origótól távolodva egyre több zérus értékkel találkozunk. A kis értékektől, melyek nem túl jelentékenyek az eljárás szempontjából, egyszerűen úgy szabadulhatunk meg, ha meghatározunk egy kvantálómátrixot, és a DCT után kapott elemeinket mind elosztjuk a kvantálómátrix megfelelő elemével. Ennek hatására bizonyos kis értékek teljesen el fognak tűnni, illetve más értékek kevesebb bit segítségével lesznek ábrázolhatók. Így a kvantálás teszi az algoritmust leginkább veszteségessé, ugyanakkor ez kínálja fel az egyik legnagyobb mértékű adattömörítést.

5. Kódolás (Coding)
   A redundáns adatmennyiség még ezután is nagy a blokkban, így egy olyan kódolási technika alkalmazása a cél, mely ezektől a redundanciáktól hatékonyan megszabadít, azonban tovább nem teszi veszteségessé az adathalmazt. Elsőként a számokat a (bitszám, értékbitek) formában alakítják át (8. táblázat), amellyel a felesleges szélességű bitábrázolást kerülik el, másrészt az adatokat az origó köré rendezik, tehát az együtthatók cikk-cakk alakban kerülnek kiolvasásra (7. ábra), valamint a sok nulla elem miatt futamhossz-kódolást alkalmaznak, melyet tovább tömörítenek Huffmann-kódolás segítségével. Az így nyert bitsorozat már alkalmas fájlba írásra.

3.1.2. A fájlformátum

A fent bemutatott tömörítési eljárás kétféle fájlformátumban terjedt el. Az egyik, a digitális kamerák és egyéb eszközök által széles körben támogatott JPEG/Exif, míg a másik az interneten fellelhető képek nagy többségének formátuma, a JPEG/JFIF. Továbbá a

7. ábra. Az együttható kiolvasási sorrendje cikk-cakkban

DCT együtthatók kiírásának sorrendjétől függően megkülönböztetünk bázisvonalas (baseline) és progresszív (interlaced/progressive) képeket is.[8] Az előbbinél a blokkok (és az azokban szereplő értékek) egymás után kerülnek letárolásra, így azokat egymás után lehet csak dekódolni, míg a progresszív esetében minden blokkból egy-egy érték kerül tárolásra, majd a következők, és így tovább. Ez az eljárás nagy fájlok letöltése esetén, már a letöltés korai szakaszában is lehetőséget ad egy előnézet megjelenítésére a teljes képről, szemben a baseline képekkel, ahol ugyan teljes részletességgel jelennek meg a blokkok, de szigorúan egymás után.

További vizsgálódásaim során a legszélesebb körben elterjedt baseline JPEG/JFIF képekre összpontosítottam, az algoritmusokat is ezekre írtam meg.

A JPEG fájlok szegmensekre vannak partícionálva, különleges marker bájtok segítségével, amit a 3. táblázatban foglaltam össze.[7] Minden rész a 0xFF majd a megfelelő marker bájt értékével kezdődik, így itt meg is kell említeni, hogy a payload (képi adat) során előforduló összes 0xFF után egy 0x00 érték kerül beszúrásra (ún. byte-stuffing művelet), hogy a dekódoláskor ezt ne lehessen marker byte-tal összetéveszteni. Ezután a header hossza, majd az adott headerre (legfontosabbak: SOI, SOF, DHT, DQT, SOS, EOI – ezek minden esetben előfordulnak) jellemző további paraméterek következnek, amelyek majd a dekódolást is lehetővé teszik. Végezetül a már említett payload az SOS marker után kezdődik.

Az általam megírt függvénykönyvtár a későbbiekben ismertetett módon csak a kvantálási tábla utáni makroblokk-értékeket fogja visszaolvasni, ehhez elegendő lesz a Huffmann-tábla (mely a futamhossz-kódolásnál használt értékeket tartalmazza) kiolvasása, mely alapján a kívánt pontra immáron az összes makroblokk visszaolvasható. Ehhez nyilvánvalóan szükséges a makroblokkok számának (továbbá a színtér-konverziónak) az ismerete. Ezeket a leíró SOF szegmens tartalmazza, az utóbbit közvetlenül, az előbbit közvetett módon a szélesség, magasság megadásával. A későbbi fejlesztési lehetőségek miatt a további headerek kiolvasását végző kódrészleteket is megírtam, így akár majd progresszív képek kezelését is meg lehet valósítani.

Marker neve	Marker hexa kódja	Leírása
SOI	0xD8	Kép kezdete
APP0	0xE0	JFIF alkalmazás szegmens
APPn	0xE1 – 0xEF	Más alkalmazás szegmens
DQT	0xDB	Kvantálási tábla definíció
SOF0	0xC0	Start of Frame (Baseline)
SOF2	0xC2	Start of Frame (Progressive)
DHT	0xC4	Huffman tábla definíció
SOS	0xDA	Start of Scan
EOI	0xD9	Kép vége (End of image)

3. táblázat. A főbb JPEG markerek

3.2. A tömörítés által okozott hibák

A JPEG-tömörítés megismerése után nyilvánvaló, hogy pontosan azokat a redundáns alsó biteket fogja az eljárás manipulálni, melyekre mi ezt megelőzően adatot rejtettünk. Így valamiféle módosítással kell élnünk az algoritmusban, hogy a tömörítés után is kielégítő valószínűséggel tudjuk a rejtett adatot visszaállítani. Célszerűnek tűnik, hogy a felsőbb biteket (azaz azokat, ahol a JPEG tömörítés már kis mértékű hibát okoz, viszont a képen szembetűnő változások még nem tapasztalhatók) alkalmazzuk ehhez. Azonban megfigyelhető, hogy a tömörítés során már kis arányú hiba is meg tudja a felsőbb biteket változtatni, például amennyiben 0x07 ($=0000\ 0111_2$) értékű az egyik színkomponens bájtja, és a tömörítés/kiolvasás során mindössze 1 eltéréssel lehet ezt visszaállítani, és így 0x08 ($=0000\ 1000_2$) értékű lesz, már akkor is 4 bit hibával kell szembenéznünk.

Erre megoldást kínálhat a pixelt alkotó bájtok adatrejtés előtti átalakítása a következőképpen: a rendekezésre álló 256 hosszú tartományt osszuk fel 2^k hosszú, ezáltal 2^{8-k} egyforma nagyságú tartományra (ahol k azt a bitet jelöli, amit adatrejtés céljára fel szeretnénk használni, szintén LSB-től, 0-val kezdve számozzuk a biteket). Ezután minden tartományban található értéket a tartomány közepére konvertáljunk át (mely a tartomány legalsó értékénél 2^{k-1}-gyel nagyobb). Végül az adatot bitenként a k bitpozíciókra rejtsük el. A módszer hatására elvben a tömörítés során injektálódó $2^{k-1} - 1$ abszolútértékű hiba még a tartományban marad, így a rejtett bit helyesen kiolvasható.
Például $k = 4$ esetén a módszer $2^k = 16$ hosszú, $2^{8-k} = 16$ darab tartományt jelent. 0000 0000-tól 0000 1111-ig terjed az első tartomány, és az átalakítás után az összes ide tartozó értéket a 0000 1000 fogja reprezentálni. Jól látható, hogy 7 nagyságú hiba még egyik irányban sem ér ki a tartományból.
A módszer hátránya, hogy a képen fura kinézetet okoz, és ezáltal hiába biztosít elvben nagyobb robosztusságot a tömörítéssel szemben, a gyakorlatban lehet, hogy könnyen szembetűnő változásokat jelent.

Az első mérések során azt vizsgáltam, hogy egyáltalán maga a tömörítés milyen hibákat

jelent a különböző módszerek során. Így teszteltem az átalakítás nélküli képeket, valamint az átalakítottakat, és mindkét esetben a jelentős tömörítési különbségeket okozó színmintavételezéssel, illetve anélkül. A módszerhez egy széles körben elterjedt, ingyenes képmanipuláló programot, az IrfanView-t használtam, mely esetén parancssori kapcsolóval szabályozható a tömörítés minősége, illetve egy .ini fájlban lehet állítani a színmintavételezéssel kapcsolatos opciókat. A méréseket így tehát ennek, és egy erre a célra írt programkódnak a segítségével végeztem el, mely 50–100% közötti tömörítési minőségekre (5–10%-os léptékben) mind a 4 esetnek megfelelően meghívta a külső programot, majd a JPEG-fájlt visszaolvasva annak „jóságát” mérte. Ehhez véletlenszerűen állítottam a k. biteket, mintha valós adatot hordozna.
A lépések tehát a következők voltak:

1. Tesztelendő BMP kép beolvasása
2. k-adik bitek véletlenszerű feltöltése, mintha adatot hordozna (=img1)

   Tartományi átalakítás (=img2)
3. i_view32.ini [JPEG] NoSampling:=1, azaz a Chroma Subsampling tiltása
4. q=50; q≤100; q:=q+5
   (a) img1 mentése q minőséggel
   `i_view32.exe img1 /jpgq=q /convert=img1_ki`
   (b) img2 mentése q minőséggel
   `i_view32.exe img2 /jpgq=q /convert=img2_ki`
   (c) Visszaolvasása img1_ki, img2_ki fájoknak, összehasonlítás az eredetiekkel (img1, img2), eredmények tárolása
5. i_view32.ini [JPEG] NoSampling:=0, 4:2:0-ás Chroma Subsampling engedélyezése
6. q=50; q≤100; q:=q+5
   (a) img1 mentése q minőséggel
   `i_view32.exe img1 /jpgq=q /convert=img1_ki`
   (b) img2 mentése q minőséggel
   `i_view32.exe img2 /jpgq=q /convert=img2_ki`
   (c) Visszaolvasása img1_ki, img2_ki fájoknak, összehasonlítás az eredetiekkel (img1, img2), eredmények tárolása
7. Eredmények képernyőre, fájlba írása további feldolgozásra

A függelékben (12, 13, 14 ábrák) láthatóak a mérések eredményei, különböző képfájlokra, változó k értékekre. Itt rendre az egyes színkomponensekbe rejtett bitek helyes visszaolvasási valószínűsége szerepel a JPEG Quality Factor függvényében. Ezekről a következő megállapításokat vonhatjuk le:

- Nagyban függ magától a képfájltól, hogy milyen valószínűséggel olvashatók vissza az adatok
- Minden esetben a színkomponensek valószínűségei G,R,B sorrendben szerepelnek. Erre magyarázat, hogy a színtér konverziós egyenletekben Y is ilyen nagyságrendű súlyozással tartalmazza ezeket (jól szembetűnő ez a 13/c és d ábrán, ahol a 4:2:0 mintavételezés miatt pontosan ez az Y komponensek által hordozott információ domínál)
- Még nagy minőségi faktor ($q \approx 70 - 80\%$) mellett is csekély, 60% körüli a helyes bitek visszaolvasása, ez természetesen gátolni fogja az adatrejtés mennyiségét
- A 4:4:4-es, illetve a 4:2:0-ás mérések között jelentős (akár 1.3-1.4-szeres) eltérések mutatkoznak a nem színmintavételezett valószínűségek javára
- Ezzel szemben a tartományi átalakított, illetve e nélküli képeken nem látszik olyan jelentős különbség, ugyanakkor a képeken jelentősebb elváltozást okoz a tartományi átalakítás

Ezek alapján célszerűnek mutatkozik a nem színmintavételezett, és nem módosított eljárásokat előtérbe helyezni. A továbbiakban ezek további javítására hibajavító kódok használatának lehetőségét fogom vizsgálni, illetve immáron konkrét adatokra is elvégzem a méréseket.

3.3. Kisebb arányú karakterhibát garantáló módszerek

Mivel láthatóan kis mértékű tömörítés is már hatalmas bithibát, és ezáltal karakterhibát eredményez, célszerű valamilyen hibajavító kódolás használata. Választásom a Hamming(8,4) kódra esett[2], mely minden 4 bitből további 4 paritásbit hozzáadásával egy 8 bites szót képez (tehát megduplázza az adat terjedelmét), ugyanakkor képes az egy- illetve kétbites hibák detektálására, valamint az egybites hibákat javítására (azaz egy SECDED – Single Error Correcting, Double Error Detecting – kód). Így egy bizonyos ($p \approx 0.672$) bithiba-valószínűség felett csökkenti a teljes karakter hibáinak valószínűségét.

	A teljes kódolt adatbyte							
	7	6	5	4	3	2	1	0
	p1	p2	d1	p4	d2	d3	d4	p8
p1	x		x		x		x	
p2		x	x			x	x	
p4				x	x	x	x	
p8	x	x	x	x	x	x	x	x

4. táblázat. A Hamming(8,4) esetén használt paritásbitek elhelyezkedése, az adatbitek lefedettsége

A módszer a különböző paritásbitekhez más és más adatbitek ellenőrzését rendeli oly módon, hogy ezáltal a fent említett hibadetektálás és -javítás egyszerűen kivitelezhető legyen. A 4. táblázat mutatja, hogy mely paritásbitek pontosan mely más bitekért is felelősek. Jól látható, hogy a paritásbitek önmagukat is lefedik minden esetben, ezáltal lehetőséget adva arra, hogy páros illetve páratlan paritást is alkalmazzunk. A további matematikai megfontolások miatt célszerű a páros mellett dönteni, azaz, hogy a paritás a lefedett egyes bitek számát párosra egészíti ki.

Maga a kódolás és ellenőrzés – a Hamming-kód lineáris volta miatt – mátrixszorzások segítségével is elvégezhető. Ehhez szükséges a G generátor (13), valamint P paritás-ellenőrző (14) mátrix.

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{pmatrix} \tag{13}$$

$$P = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{14}$$

Ezek segítségével egy adott $r = (a_1, a_2, a_3, a_4), a_i \in T_1$ bitnégyeshez (ahol $T_1 = \{0, 1\}$ kételemű test) tartozó kódszót a G generátormátrix segítségével kaphatjuk meg: $c = rG$ az eredmény egy vektor, melyet szintén a T_1 testen kell tekinteni. Hasonlóképpen egy kódszó elllenőrzése a P mátrix segítségével: $w = Pc^T$ módon történik, a w vektor a null-vektor, amennyiben nincs hiba, egybites hiba esetén az eredmény a hibás bitnek megfelelő oszlopával egyezik meg a P mátrixnak, mivel $w = Pc^T = P(h+e)^T = P\left(h^T + e^T\right) = Pe^T + Pc^T = Pe^T$ (ahol e az injektálódó hibát modellezi).

A felhasználás során a mátrixműveleteket célszerű elkerülni, hiszen jelentősen lassítják a számítások idejét. Ezért mind a kódolás mind a dekódolás során ezeket egy előre kiszámított és letárolt const tömbből hívom vissza. A dekódolás során az egybites hibák esetén már a javított értékkel, kétbites hibák esetén egy hibajelző -1 értékkel tér vissza a Hamming-dekódoló függvény, jelezve ezáltal a további felhasználás számára a szükséges információkat. A használt kódszavak táblázatos formában a 6. táblázatban található.

3.3.1. A különböző hibajavító kódolások hatékonysága

Amennyiben nem alkalmazunk hibajavító kódolást, és a helyes biteket független p valószínűségűnek tételezzük fel, egy 8 bites karakter helyes dekódolásának valószínűsége $P(\alpha)$ (15), ami nagyon gyorsan csökkenő tendenciát mutató függvénnyel ($p = 0.9$ esetén is már csak $f(0.9) \approx 0.43$) jellemezhető. A valószínűségfüggvény a 8. ábrán feketével jelölt.

$$P(\alpha) = p^8 \tag{15}$$

Hamming(8,4) kódolás használata esetén egy karakter két 4 bites adatra bontható, melyek egyenként 8-8 bites kódszóval kerülnek kódolásra. Ahhoz, hogy ezt helyesen dekódoljuk, mindkét kódszó esetén maximum egy bithiba megengedett. Hasonlóan a helyesen detektált biteket függetlennek, és p valószínűségűnek tételezzük fel. A hibátlan 8-as bitsorozat szintén p^8 valószínűségű, ellenben az egybites hibákat tartalmazó esetek (melyek valószínűsége $8p^7(1-p)$) is kedvezőek. Így a teljes, 4 adatbitre (8 hosszú kódszóra) vonatkozó valószínűség $p^8 + 8p^7(1-p)$. Mivel a teljes 8 bites karakterhez két ilyen 4 bites adatblokk helyes detektálása szükséges, így a karakter helyes dekódolási valószínűsége Hamming(8,4) alkalmazása esetén $P(\beta)$ (16), amely a 8. ábrán kékkel jelölt függvény. Jól látható, hogy egy valószínűségi érték felett (a $p^3(8-7p) = 1$ feltételből ez a határ $p \approx 0.672$) a Hamming-kódolás jobbnak bizonyul, mint a hibajavító kódolás nélküli módszer. Ez alatt az érték alatt azonban a sok paritásbit miatt már rosszabb valószínűségű lesz a teljes adat helyreállítása.

$$P(\beta) = \left[p^8 + 8p^7(1-p)\right]^2 = p^{14}(8-7p)^2 \tag{16}$$

Végezetül egy harmadik módszert is alkalmaztam, mely a Hamming-kódolást többszörözve használja fel, ugyanis egy pixel mindhárom színkomponensébe ugyanazt az adatot rejti. Így a dekódolás során három kódszó is az algoritmus segítségére áll az adat helyreállítására, és ezek közül egy többségi szavazásos módszerrel próbálja meghatározni a helyeset.
Amennyiben nem lenne egyik érték részére sem döntés, az előző fejezetben megismert valószínűségek ismeretében a színeken belüli prioritások sorrendje: G, R, B.
Ahhoz, hogy ennél a módszernél a valószínűséget kiszámoljuk, egy összes esetet vizsgáló 7. táblázatot kell készíteni. Itt az értékek jelentése:

- 1: A Hamming-dekódolás alapján az adott színben levő bitnégyes helyesen kerül azonosításra (azaz maximum 1 bit hiba)
- 0: A Hamming-dekódolás során 2 bites hiba szerepel, azaz az algoritmus számára a hiba ténye azonosított
- -1: A dekódolás során olyan 3 vagy több bites hiba merült fel, melyet az algoritmus csak hibásan tud azonosítani (azaz egy konkrét értéket határoz meg, és biztos benne, hogy ez helyes, pedig valójában nem)

Így ezek alapján meghatározhatók azok a kedvező esetek, amikor a módosított Hamming-eljárás után a megfelelő bitnégyest kapjuk vissza. A táblázat alapján láthatóak a kedvező (13 darab), és kedvezőtlen (12 rossz detektálás, 1 nem eldönthető, és 1, ami függ az értékektől) esetek. Amennyiben ezek közül csak a 13 szerencséset vesszük számításba, illetve a színkomponenseken belüli helyes bitdetektálásokat azonos valószínűségűnek tételezzük fel, hasonlóan meghatározható a teljes valószínűség (bár ekkor egy alsó becsléssel fogunk élni). Továbbá jelentse rendre p_{-1}, p_0, p_1 a hibás, semleges, helyes bitnégyes-detektálási valószínűségeket (az előzőek alapján $p_1 = p^7(8-7p)$; $p_0 = 28p^6(1-p)^2$,

mivel ekkor 6 helyes, 2 hibás bit van, és összesen $\binom{8}{2} = 28$ különböző ilyen eset van; $p_{-1} = 1 - p_1 - p_0$)

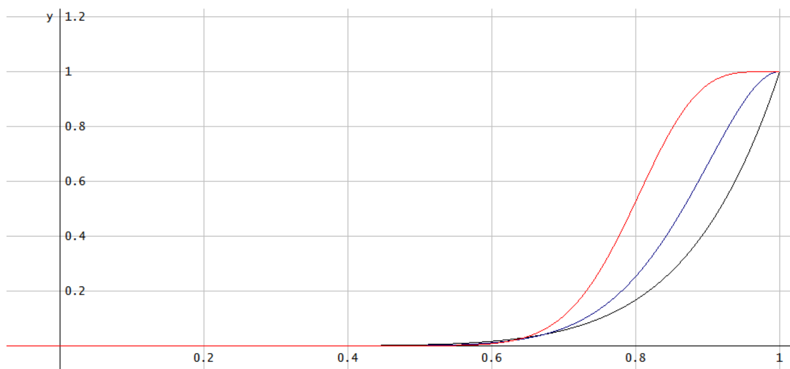
- {-1,0,1} alakú: itt 3 kedvező eset, azaz $P_a = 3p_{-1}p_0p_1$
- {-1,1,1} alakú: szintén 3 eset: $P_b = 3p_1^2p_{-1}$
- {0,0,1} a 3 eset: $P_c = 3p_0^2p_1$
- {0,1,1}: $P_d = 3p_0p_1^2$
- {1,1,1}: $P_e = p_1^3$

Ezeknek az eseteknek az összegzéséből kaphatjuk meg a teljes valószínűséget, és mivel itt szintén csak 4 bit helyes detektálásáról beszélünk, így a teljes bájt helyes detektálásának valószínűsége $P(\gamma)$ (17), amely a szóban forgó 8. ábrán a pirossal jelölt függvénygörbe.

$$P(\gamma) = [P_a + P_b + P_c + P_d + P_e]^2 \tag{17}$$

Így ezen valószínűségek ismeretében leginkább az utolsó módszer sikerében reménykedhetünk, bár még ennek a módszernek is a 80%-os karakterjósághoz legalább ilyen mértékű bitjóság szükséges. Ezeket pedig a már ismert 13, 12 ábrák alapján csak 90% vagy annál nagyobb tömörítés-jósági tényező mellett várhatjuk.

A mérések ezt megerősítik. Míg 95-100% Quality Factor mellett a rejtett szöveg teljes mértékben visszanyerhető, ennek csekély megváltoztatása is az adatok visszaállíthatóságának rohamos csökkenését jelenti (már 90% és 4 bit esetén – ami kis tömörítés, és majdnem látható adat – is csupán 15-20%-a olvasható vissza a karaktereknek). Néhány próbálkozás után kijelenthető, hogy ezen módszer nem csak rossz képminőséget, kevés rejthető adatot biztosít, de még azt is csak különleges körülmények között képes visszaadni. Habár a hibajavító kódolások használata megoldást jelenthetett volna, mégsem teljesítette a vele szembeni elvárásokat.



8. ábra. Különböző hibajavító kódolások alkalmazásában rejlő lehetőség

4. Adat rejtése JPEG-tömörített hordozókba

Az előző fejezet alapján nyilvánvaló, hogy habár csekély keresztmetszettel, de lehetséges az adatot rejtő képek tömörítése, és az adatok visszaolvasása (egy bizonyos valószínűséggel), ennek mértéke egyáltalán nem közelíti meg a lehetőségeinket tömörítetlen képfájlok esetén. A cél, hogy valamilyen módon ötvözzük a kis kimeneti fájlméret, és viszonylag nagy mennyiségű rejtett adat elvárásokat. Erre kínál megoldást a JPEG tömörített fájlokon végzett adatrejtés.

4.1. Az adatrejtés elve

Az előző fejezetben ismertetett JPEG-szabvány rengeteg előnye mellett kínál egy hátrányt is: a képen nem képes „szelektálni", azaz a tömörítés minősége, mértéke azonos lesz minden tartományban. Így ahhoz, hogy kellő pontossággal tudjuk megjeleníteni a számunkra fontos nagyfrekvenciás részeket (pl. egy portré esetén haj, szakáll stb.) nagy kimeneti minőségfaktort kell beállítani. Ezáltal viszont a képnek bizonyos kevésbé jelentős részei felesleges pontossággal lesznek ábrázolva. Így – hasonlóan a tömörítetlen képfájlokhoz – célunk ezeknek a redundáns adatoknak a megkeresése, és kicserélése az elrejtendő adatra. Ugyanakkor ügyelni kell arra, hogy olyan szinten módosítsuk az adatokat, ami után már nincs veszteséges tömörítési eleme a JPEG-szabványnak. Ez közvetlenül az utolsó, entrópia kódolás lépés előtt lehetséges, hiszen az ezt megelőző kvantálás lépés mindenképpen veszteséges (lehetséges lenne a csupa '1'-es mátrixszal kvantálni, ekkor azonban nagyságrendekkel megnő a fájlméret). Így a továbbiakban azt fogom vizsgálni, hogy ezen a ponton milyen formában lehetséges a redundáns biteket megtalálni.

Mivel ingyenes függvénykönyvtárat – mely a kérdéses ponton lehetővé teszi az adatok módosítását nem találtam – így ezt is megvalósítottam, mely jelenleg a legjobban elterjedt baseline, 4:4:4-es illetve 4:2:2-es színblokk átlagolt képeket támogatja. Az említett ponton (a kvantálás után) a makroblokkok már a fájlba írásra kész értékeket tartalmazzák, ezek tehát YCbCr színtérben, esetleg színblokk átlagolt, diszkrét koszinusz transzformált, kvantált értékek. Fontos ismét megemlíteni, hogy a koszinusz transzformáció a blokk legnagyobb részét az alacsonyfrekvenciás komponensekbe viszi át, a nagyfrekvenciás értékek csak néhány pixelt (a már említett haj, vonalak, rácsok, egyéb finom részletek) módosítják. Így óvatos változtatásuk az emberi szem számára még nem lesz látható. A mérések során változtatható értékeket a következő paraméterek határozzák meg:

- $Y_{min}, C_{b_min}, C_{r_min}$
  Elsősorban azt a pontot kell megszabnunk, ahol még egyáltalán nem történik módosítás a makroblokkokban. Ezt határozzák meg a *min* értékek, melyek azt a sorszámot hordozzák, amelyik elemét a makroblokknak még nem lehet módosítani (azaz pl. $Y_{min} = 2$ esetén az Y komponensű makroblokkokban az első két elem változatlan marad, és csak a 3-tól kerülnek esetleges módosításra). Ezzel kapcsolatban fontos megjegyezni, hogy az Y értékek hordozzák a szürkeárnyalatos (fényesség) információt, így fontosságuk miatt célszerű az Y_{min} értéket nagyobb értékűre választani (általában

legalább 4-8), ugyanakkor C_{b_min} és C_{r_min} egészen kicsi is lehet (érdekes módon akár 2 is). Illetve, mivel a blokk első eleme minden esetben a legjelentékenyebb (DC) érték, amit célszerű nem módosítani, így az állítható tartomány a programban is 1-ről indul.

- Védett bitek (b)
  Ezzel az értékkel azt szabályozhatjuk, hogy milyen mélyen kerüljön átírásra egy adott makroblokk-érték. Egészen pontosan a felső b bit nem kerül átírásra, kivéve, ha csak 1 darab bitből áll a szám. Ennek a célja a kép 'invertálódásának' elkerülése; a felső bitek átírása (invertálása) ugyanis a 8. táblázat alapján láthatóan magának a makroblokk-beli számértéknek az invertálását jelenti. Ez a képnek egy fura, töredezett megjelenést fog adni, amely elkerülendő.

- Offset
  Az adatok elrejtése szintén egy téglalap alakú területre kerül, viszont ezeknek határait a makroblokkok szabják meg. Így az offset sem lehet bármilyen érték, célszerű egész makroblokkok számában megadni.

- Szélesség, magasság
  Az előző alapján a terület szélessége és magassága szintén egész makroblokk darabszámban van megadva.

Magának az adatnak az elrejtése hasonlóan a tömörítetlen képekhez egy headerben történik, melyek elemei is nagyjából kölcsönösen megfeleltethetők egymásnak. A JPEG képekre használt header a 5. táblázatban látható, első eleme szintén egy marker, mely jelen esetben „FILE”, amely talán nem szerencsés abból a szempontból, hogy szövegfájlok rejtése esetén maga az adat is tartalmazhatja ezt a kifejezést, azonban annak valószínűsége, hogy első bitje pontosan egy makroblokk első bitjére esik (a kiolvasás szempontjából ez lesz fontos), csekély. Ezt követi a befoglaló téglalap szélessége, mely makroblokk darabszámban van megadva, ahogyan az már említésre került. Ezután a rejtendő fájl nevének hossza 1 bájton, majd magának a fájlnak a mérete, 4 bájton (mely a maximális 4.3 GB fájlmérettel bőséggel elegendő), ezután a fájlnév maga és az adat. A fájlnév tárolásának oka, hogy magából az adatból annak formátuma nem ismert, így legegyszerűbb a nevével (és annak kiterjesztésével) azonosítani tartalmát.

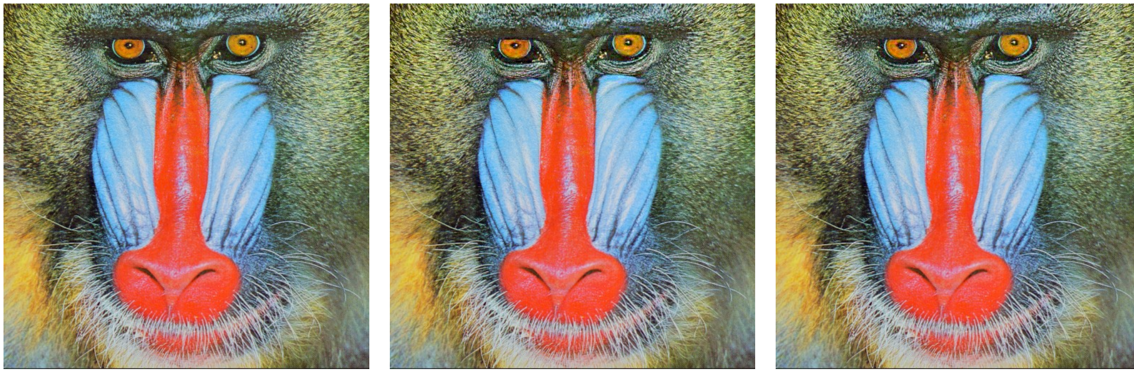
SoF marker "FILE"	Szélesség (Makroblokk darab)	Fájlnév hossza (n)	Fájl mérete (f)	Fájlnév	Fájl tartalma
0x46494c45	Little-endian		Little-endian		
4 bájt	2 bájt	1 bájt	4 bájt	n bájt	f bájt

5. táblázat. A header formátuma JPEG esetén

4.2. A mérések

A következőkben a már megismert tesztképek JPEG-tömörített változataira fogok elrejteni más fájlokat (képeket, hiszen azok egyrészt nagyobb mennyiségű adatot tartalmaznak, illetve az algoritmus helyes működése könnyen leellenőrizhető a kiolvasás után, egyszerű „szemrevételezéssel". A továbbiakban a paraméterek rövidítésére a $\{Y_{min}, C_{b_min}, C_{r_min}, b\}$ vektorjellegű rövidítést fogom használni. Egyaránt vizsgálom a 4:4:4-es, tehát nem színmintavételezett, valamint a 4:2:0-ás mintavételezett képeket, és a közöttük rejlő különbségeket.

Elsőként a mandrill 4:2:0-ás tömörített változatába (mely így kb. 126 KB) helyeztem el egy portrét (a kép felső felére, {4,1,1,2}-es paraméterekkel), mely 9 KB méretű állomány volt, így kb. $9k/63k \approx 14\%$ adatrejtési arányt elérve. A módosítás ebben az esetben még meglátszott a képen, így egy {4,3,3,2}-es változatot is kipróbáltam, mely már jobb eredményeket mutatott, ugyanakkor csupán kissé nagyobb területen. Így ebben az esetben a teljes képre vonatkoztatva 14%-os adatrejtés még nem látható (9. ábrasor).



(a) A mandrill, immáron JPEG-tömörítve, 4:2:0-ás Chroma Subsampling (b) Elrejtett adattal {4,1,1,2} (c) Ugyanarra az adattömbre, {4,3,3,2}

9. ábra. JPEG-rejtés tesztképei különböző paraméterekre, 4:2:0-ás színmintavételezésre

A következőben egy erdő 4:4:4-es JPEG-tömörített változatával kísérleteztem. Különösképpen a paraméterek növelésével (elsőként {5,3,3,2}, majd fokozatosan egészen {10,8,8,3}-ig) az észrevehetőség nem csökkent, ami azzal magyarázható, hogy míg ebben az esetben az átlagolás elmaradása miatt minden makroblokk egy az egyben megfeleltethető egy 8x8-as pixelcsoportnak, a változtatások is jobban láthatóak lesznek. A másik esetben pedig az átlagolás az általunk a képre injektált zajt is eloszlatja, így az kevésbé lesz szembetűnő. Azaz általánosságban elmondható, hogy a nem mintavételezett képek potenciálisan nagyobb mérete, és adatbefogadási képessége pontosan ellenkező hatást ér el, mint az várható. Ezt jól alátámasztja az erdő újratömörített változata, mely így már jóval kevésbé észrevehető módon képes adatot hordozni (az összehasonlítás elvégezhető a 10. ábra alapján (a tömörített rejtés nagyméretű képei a függelékben 25-30-ig találhatók).

(a) Az erdő, JPEG-tömörítve, No Chroma Subsampling

(b) Elrejtett adattal {5,3,3,2}

(c) Ugyanarra az adattömbre, {10,8,8,3}

(d) 4:2:0-ás Chroma Subsamping engedélyezésével, {5,3,3,3}

10. ábra. A színmintavételezés tiltásának hatása JPEG-rejtés esetén

5. Az algoritmusok biztonsága, felhasználása

Az algoritmusok elvi megismerése után a felhasználásukat, és mivel valamilyen szempontból titkosításról beszélünk, a biztonságukat vizsgálom meg.

5.1. Alkalmazási területek

Az alkalmazások sorában kézenfekvő első helyre sorolni valamilyen titkos információ elrejtését – akár csak tárolási, de továbbításra szánt rejtett üzenetként is –, és az ehhez kapcsolódó felhasználásokat. A módszernek ilyenkor csupán egy hátránya adódik: a paraméterek megosztása (offset, mélység, védett bitek stb.) nehézkes a két fél számára, viszont optimalizációs szempontból célszerű lenne ezeket megválasztani mégis a rejtés során. Továbbá látni fogjuk, hogy a paraméterek ismerete nélkül sem túl nehezen megtalálható a rejtett üzenet. Így jogosnak tűnhet, hogy a paraméterek megosztási problémájával nem törődnek – nyilván próbálják titokban átjuttatni –, hanem az adatot kódolják valamilyen modern, nyilvános kulcsú, vagy egyéb módszerrel. Ekkor még ha fény is derül arra, hogy a kép hordoz plusz információt, annak megfejtése jelen állás szerint valós időben nem teljesülhet. A felhasználás ilyen módjára több hivatkozás is található, legismertebb és egyben legvitatottabb a 9/11 támadások szervezésére utaló hírek csoportja[12][13][14]

További alkalmazási terület lehet törékeny vízjelek elhelyezése képekben. Ekkor egy szöveget vagy képet/fájlt – mely köthető az adott kép készítőjéhez – rejthet el abban, így később láthatóvá teheti, hogy hozzá kapcsolódik az az adott kép. Hátránya, hogy gyakorlatilag teljesen érzékeny az újratömörítésre mindkét módszer, így ezután a vízjel is el fog tűnni (ezért törékeny, ha ez nem befolyásolná, robosztus vízjelről beszélhetnénk)[1]

5.2. A rejtett adatok biztonsága

A legfontosabb szem előtt tartandó tény, hogy amennyiben a kép eredeti változatával rendelkezik valaki, akkor annak segítségével egy egyszerű összehasonlítás eredményeképpen megállapítható, hogy van-e olyan terület, ahol történt bitek módosítása. Ezután már az adott területre véges számú próbálkozással felfedhető az adat (már amennyiben nem titkosították). Éppen ezért nagyon fontos a közismert, elterjedt, interneten is széles körben fellelhető képek kerülése. A legjobb megoldás készíteni egy teljesen eredeti képet hordozónak minden esetben.

Továbbá a kriptográfiában jól ismert állítást, miszerint egy eljárás biztonsága nem függhet az algoritmustól, csupán a kulcstól, itt is megvizsgálhatjuk. Amennyiben valaki rendelkezik az algoritmusok felépítésével (frame szerkezete, paraméterek), akkor véges számú próbálkozás (ami nagyságrendjében is kicsiny) után megtalálhatja, amit keres. Például a tömörítetlen esetben változhatható az offset és a bitmélység, azaz $n \times m$-es kép esetén ez kb. $8nm$ esetet jelent (az adat hosszától függően valamivel kevesebb). Tesztelés céljából egy 50×50-es területet vizsgálva kb. 5-10 mp alatt lehet a legszélsőségesebb esetben

az adatra bukkanni, így még egy hatalmasnak tekinthető 5000×5000-es kép esetén is modernebb számítási képességű eszközökkel (legalább 10x gyorsabb) is ez bőven egy óra alá szorítható. A jellemző pár száz pixel széles/magas képekre pedig már egy egyszerű számítógéppel is percekbe kerül.

Az eddigiek alapján tehát látható, hogy egyrészt ritka képeket kell választani, másrészt a képre már titkosított adatot kell rejteni. Azonban még ez sem jelent biztonságot a különböző statisztikai elemzésekkel szemben. Legyegyszerűbb az elsőrendű (hisztogramszerű) statisztikai vizsgálat[3], mely során az egyes színek gyakorisága kerül elemzésre. Jelöljük egy adott i szín rejtés előtti gyakoriságát n_i, az ez utánit n_i^*-gal. Ha 1 bites LSB rejtést vizsgálunk, és feltesszük hogy $n_{2i} > n_{2i+1}$, azaz a $2i$-edik szín gyakorisága nagyobb, mint az őt követőé, és az üzenetben egyenletes eloszlású bitek szerepelnek, nagy valószínűséggel fog a $2i$-edik és $2i+1$-edik szín eloszlása egymáshoz közelíteni, így a $|n_{2i} - n_{2i+1}| \geq \left|n_{2i}^* - n_{2i+1}^*\right|$ egyenlőség fennállni. Így tehát az egyenletes eloszlású rejtett adatbitek csökkentik az egymást követő színek gyakoriságainak különbségét. Ez fennáll mind tömörítetlen képek, mind JPEG-tömörített képeken a makroblokk-elemek alsó bitjeire történő rejtéskor. Ezzel a módszerrel történő ellenőrzéskor a szomszédos színek valószínűségeit átlagoják, és ezt összehasonlítják egy rejtett adatot nem hordozó képről készült eloszlással. Ezt az elvet használja az OutGuess[15] program is. A módszer ellen védekezni kevés adat nagy területen való szétszórásával lehet, ahogyan a [3][5] források is megemlíti.

További elemzési lehetőségeket kínál a magasabbrendű statisztikai elemzések szupport vektor gépekkel (SVM) történő végrehajtása[4]. Itt az eloszlások adatot tartalmazó és nem tartalmazó képekre történő meghatásozása után ezeket a legegyszerűbb esetekben lineárisan szeparábilis és nem szeparábilis SVM-k segítségével lehet osztályozni. Bonyolultabb esetekre nemlineális SVM-ek alkalmazása szükséges. A betanítási folyamat után egy képről már megállapítható, hogy eloszlása melyik osztályba tartozik, így hordoz-e rejtett információt. Az eljárás részletes ismertetése Siwei Lyu és Hany Farid cikkében[4] található.

Az eljárások előnyeinek, és immáron hátrányainak ismeretében az alkalmazás függvényében választható meg a pontos felhasználás, illetve a paraméterek, melyek változtatását sokkal jobban biztosítja a jelenleg interneten fellelhető[16][17] alkalmazásokkal szemben. Ezen mérések elvégzéséhez hasonlóan érdekes lehet más adatrejtési irányok, így például hangfájlok vizsgálata. Továbbá célomnak tekintem a JPEG-konverziós függvénykönyvtár teljessé tételét, így az algoritmusok kiterjesztését progressive és más JPEG típusokra is, illetve általánosabb felhasználásokra. Valamint ezzel párhuzamosan a keretprogramokat is szeretném egy megbízhatóan, gyorsan működő változatig fejleszteni.

Hivatkozások

[1] Tamás, Virasztó: *Titkosítás és adatrejtés — Biztonságos kommunikáció és algoritmikus adatvédelem*, NetAcademia Kft., 2004

[2] Moon, Todd K.: *Error correction Coding – Mathematical Methods and Algorithms*, Wiley, 2005

[3] Provos, Niels — Honeyman, Peter: *Hide and Seek: And Introduction to Steganography*, May/June, 2003. (`http://www.citi.umich.edu/u/provos/papers/practical.pdf`, megnyitva: 2011. október 25.)

[4] Lyu, Siwei — Farid, Hany: *Detecting Hidden Messages Using Higher-Order Statistics and Support Vector Machines*, (`http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.57.6352&rep=rep1&type=pdf`, megnyitva: 2011. október 25.)

[5] Orebaugh, Angela D.: *Steganalysis: A Steganography Intrusion Detection System*, `http://www.securityknox.com/Steg_project.pdf`

[6] Millar, Campbell Dr.: *Digital Signal Processing 2*, Oktatási segédanyag, (`http://eng.moodle.gla.ac.uk/file.php/272/LectureNotes2011.pdf`, megnyitva: 2011. szeptember 18.)

[7] Amin, Viren Dr. (Iowa State University): *JPEG file layout and format*, Oktatási segédanyag (Digital Image Processing tárgyhoz), 2005, (`http://class.ee.iastate.edu/ee528/Reading%20material/JPEG_File_Format.pdf`, megnyitva: 2011. szeptember 9.)

[8] Hamilton, Eric (C-Cube Microsystems): *JPEG File Interchange Format*, 1992. szeptember 1., (`http://www.jpeg.org/public/jfif.pdf`, megnyitva: 2011. szeptember 18.)

[9] Murray, James D. — Van Ryper, William: *Encyclopedia of Graphics File Formats, 2nd Edition, Chapter 09.*, May 8, 1996., (`http://www.fileformat.info/mirror/egff/ch09_06.htm`, megnyitva: 2011. szeptember 18.)

[10] Leong, Jennifer: *Number of Colors Distinguishable by the Human Eye*, (`http://hypertextbook.com/facts/2006/JenniferLeong.shtml`, megnyitva: 2011. október 25.)

[11] *Secret Writing* (`http://www.foia.cia.gov/CIAsOldest/Secret-writing-document-three.pdf`, megnyitva: 2011. október 12.)

[12] Schneier, Bruce: *Terrorists and steganography*, Internetes újságcikk, 2001. szeptember 24., (`http://www.zdnet.com/news/terrorists-and-steganography/116733`, megnyitva: 2011. szeptember 9.)

[13] McCullagh, Declan: *Bin Laden: Steganography Master?*, Internetes újságcikk, 2001. július 2., (http://www.wired.com/politics/law/news/2001/02/41658, megnyitva: 2011. szeptember 9.)

[14] Kelley, Jack: *Militants wire Web with links to jihad*, Internetes újságcikk, 2002. október 7., (http://www.usatoday.com/news/world/2002/07/10/web-terror-cover.htm, megnyitva: 2011. szeptember 9.)

[15] Provos, Niels: *OutGuess*, (http://www.outguess.org/, megnyitva: 2011. október 25.)

[16] QuickCrypto: *QuickStego - Free Steganography Software*, (http://quickcrypto.com/free-steganography-software.html, megnyitva: 2011. október 25.)

[17] *Steghide*, (http://steghide.sourceforge.net/index.php, megnyitva: 2011. október 25.)

A. Táblázatok

Adat		Kódszó	
Decimális	Bináris	Bináris	Decimális
0	0000	0000 0000	0
1	0001	1101 0010	210
2	0010	0101 0101	85
3	0011	1000 0111	135
4	0100	1001 1001	153
5	0101	0100 1011	75
6	0110	1100 1100	204
7	0111	0001 1110	30
8	1000	1110 0001	225
9	1001	0011 0011	51
10	1010	1011 0100	180
11	1011	0110 0110	102
12	1100	0111 1000	120
13	1101	1010 1010	170
14	1110	0010 1101	45
15	1111	1111 1111	255

6. táblázat. A Hamming(8,4) kódszavai

R^{+}	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	0	0	0	0	0
G^{++}	-1	-1	-1	0	0	0	1	1	1	-1	-1	-1	0	0	0
B	-1	0	1	-1	0	1	-1	0	1	-1	0	1	-1	0	1
	-1	-1	-1	-1	-1	-1	?	+1	+1	-1	-1	-1	-1	0	+1

R^{+}	0	0	0	1	1	1	1	1	1	1	1	1
G^{++}	1	1	1	-1	-1	-1	0	0	0	1	1	1
B	-1	0	1	-1	0	1	-1	0	1	-1	0	1
	+1	+1	+1	-1	-1	+1	+1	+1	+1	+1	+1	+1

7. táblázat. A módosított Hamming kódolás esetei

Kód	Méret	Értékbitek		Eredeti érték	
00	0			0	
01	1	0	1	-1	1
02	2	00,01	10,11	-3,-2	2,3
03	3	000,001,010,011	100,101,110,111	-7,-6,-5,-4	4,5,6,7
⋮	⋮	⋮		⋮	
0A	10	00 0000 0000,...	..., 1111 1111 11	-1023,...,-512	512,...,1023
0B	11	000 0000 0000,...	..., 1111 1111 111	-2047,...,-1024	1024,...,2047

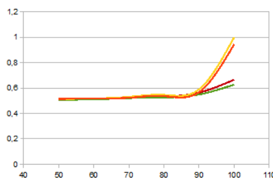
8. táblázat. A DC-n kívüli értékek konvertálása a makroblokkok futamhossz-kódolásához

B. Képek

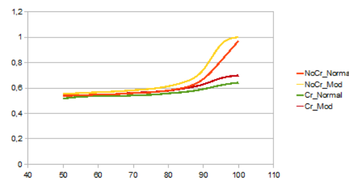
(a) 3 bites rejtés még nem látható

(b) 4 bites rejtés már látható

11. ábra. Szürkeárnyalatos kép használata tömörítetlen módon, szöveg rejtésére (rejtett adat a bal karon látható)



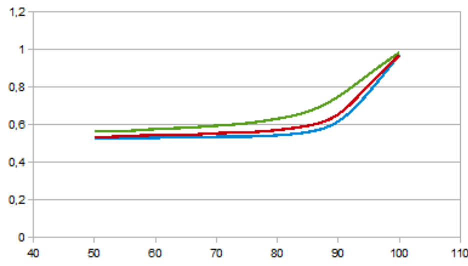
(a) k=3 esetén még nagy a hiba aránya



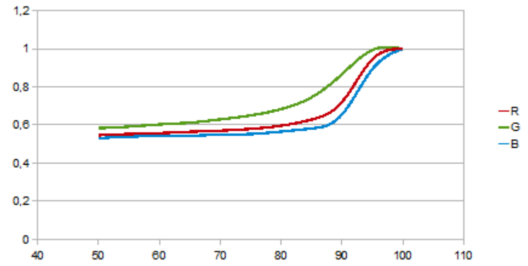
(b) k=4 esetén is lasabb a növekedés a vártnál

(c) Az elemzett kép

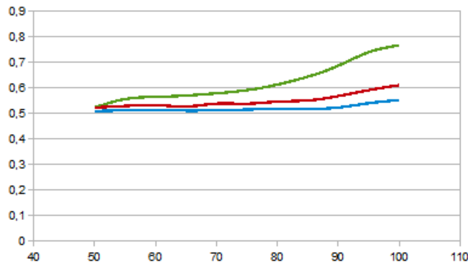
12. ábra. Egy kép viselkedése JPEG-tömörítéskor, k=3 és k=4 értékekre



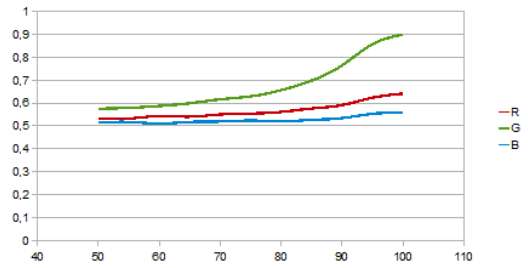
(a) No Chroma Subsampling, normál kép



(b) No Chroma Subsampling, tartományi átalakított kép

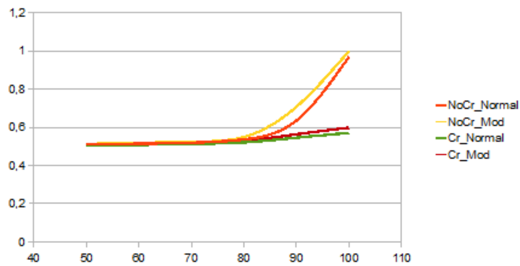


(c) 4:2:0 subsampling, normál

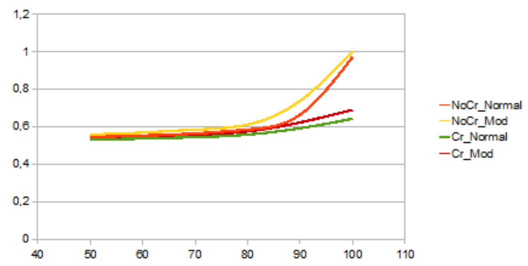


(d) 4:2:0 subsampling, tartományi álalakított

13. ábra. Az előző mérések eredményei részletesen, a 4 esetre



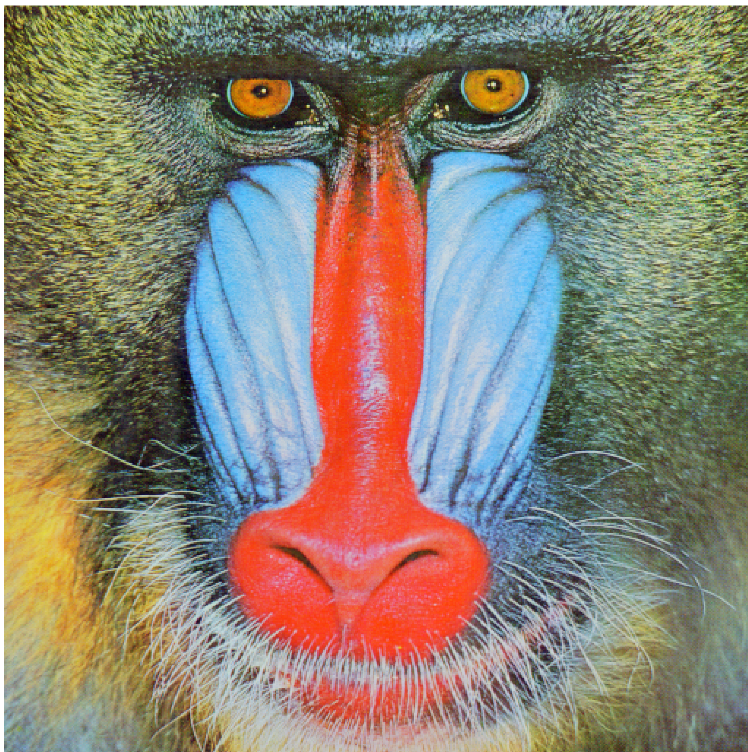
(a) A már ismert mandrill



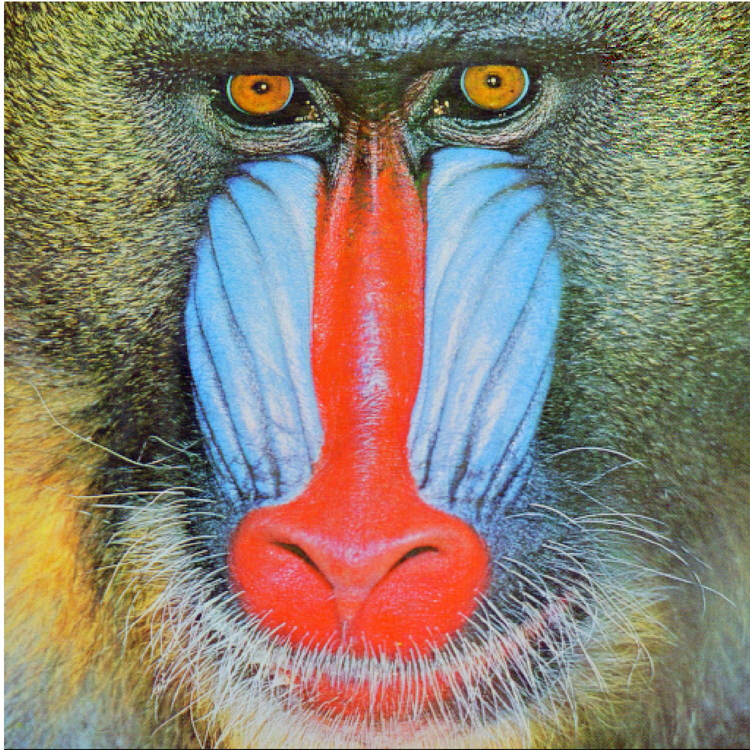
(b) És az erdő

14. ábra. Két további kép mérése

15. ábra. Az eredeti mandrill



16. ábra. Az 5 bites rejtés

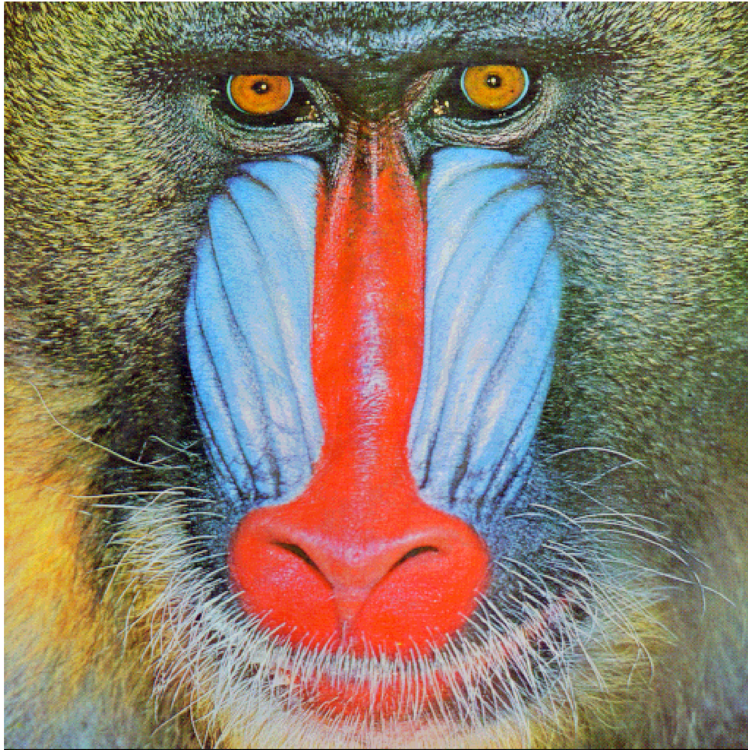


17. ábra. A 6 bites rejtés

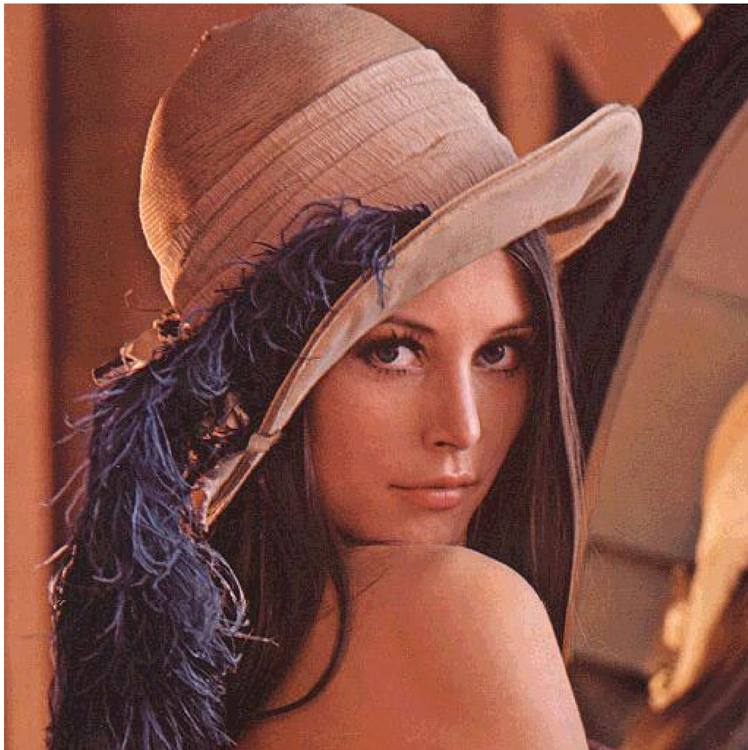
18. ábra. Az eredeti Lenna

19. ábra. A 3 bites rejtés

20. ábra. A 4 bites rejtés, már látható



21. ábra. Kép a képben módszerkor, 4 bites rejtés esetén a hordozó minősége



22. ábra. Kép a képben, 4 bit, a rejtett kép minősége

23. ábra. Kép a képben módszer, az erdő eredetije

24. ábra. Kép a képben módszer, az erdő 4 bites adattal

25. ábra. A mandrill JPEG-tömörített eredetije

26. ábra. 9k rejtett adattal, $\{4,1,1,2\}$-es rejtés

27. ábra. A rejtés megváltoztatása {4,3,3,2}-re

28. ábra. Az erdő {10,8,8,3}-as esetben, No Chroma Subsampling

29. ábra. Az erdő {5,3,3,2}-es esetben, No Chroma Subsampling

30. ábra. Az erdő {5,3,3,3}-as esetben, immáron 4:2:0-ás színmintavételezéssel