EÖTVÖS LORÁND TUDOMÁNYEGYETEM

TERMÉSZETTUDOMÁNYI KAR

HIBAJAVÍTÓ KÓDOK ÉS ALKALMAZÁSAIK

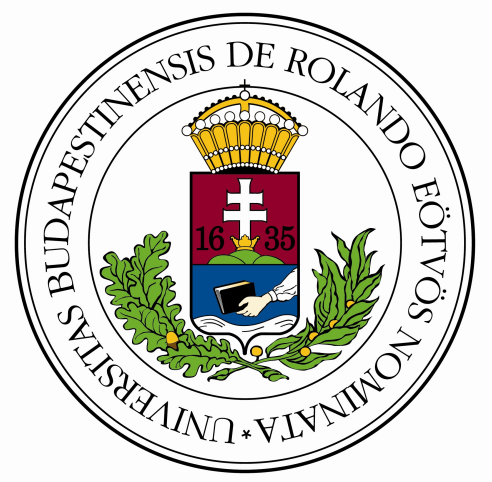
SZAKDOLGOZAT

Nagy Eszter

Matematika BSc

Matematikai elemző szakirány

Témavezető: Szőnyi Tamás



Budapest

2023

Absztrakt

A kódelmélet egyik meghatározó területe a hibajavító kódok csoportjával foglalkozik, amelynek mélyebb megértéséhez szükség van algebrai ismeretekre. Ezek közül a szakdolgozatom témájának a Reed-Solomon hibajavító kódokat választottam, amelyben a matematikai hátterét felépítve lépésről lépésre mutatom be egy üzenet kódolását és dekódolását példákon keresztül. Továbbá szemléltetek egy Python programnyelven megvalósított szimulációt, amely egy üzenetet kódol, hibát kap majd dekódol.
Az első fejezetben általánosan ismertetem a problémát, hogy egy kis bevezetőt kapjunk a témáról. A második fejezetben még a mély matematika bevezetése nélkül, nagy vonalakban kifejtem az üzenetküldés útját. A harmadik fejezetben a kód alapjául szolgáló véges testek matematikáját részletezem. A negyedik fejezetben a lineáris kódokat mutatom be, amely egy kis felvezetése már a Reed-Solomon kódoknak. Az ötödik fejezetben már részletesen mutatom be egy üzenet kódolását. A hatodik fejezetben ezen kódolt üzenet sérülését és annak jellemzőit írom le. A hetedik fejezetben a hibás kódszó dekódolását fejtem ki. Végül a nyolcadik fejezetben szimulálok egy üzenetküldést, annak megsérülését majd javítását, valamint bemutatom a programkódomat, hogy miként hajtja végre az algoritmust.
Még a témával kapcsolatos alapvető ismeretekkel semleges olvasó is teljes körű betekintést és könnyen érthető információkat szerezhet a dolgozat végére, azonban aki jártasabb a kódelmélet világában, annak is tartalmazhat új információt, hiszen egy kódot többféleképpen is lehet implementálni.

Köszönetnyilvánítás

Szeretném kifejezni szívből jövő köszönetemet a konzulensemnek, Szőnyi Tamásnak, aki szakértelmével és segítőkészségével iránymutatást adott a szakdolgozatom elkészítéséhez. Köszönöm, hogy a kérdéseimmel bármikor bizalommal fordulhattam Önhöz, tanácsaival és hibáim javításával nagy segítségemre volt.

Köszönettel tartozom továbbá Tóth Benjáminnak, aki szintén fontos szerepet játszott ezen utazásom során. Építő gondolatai mindig inspiráltak a nehéz pillanatokban.

Végül de nem utolsó sorban szeretném hálásan megköszönni az Eötvös Lóránd Tudományegyetem Természettudományi Kara valamennyi oktatójának azt a lelkiismeretes munkáját, amivel a hallgatók képzését, tanulását és a mély matematikai tudás megszerzését támogatták.

Köszönöm szépen még egyszer mindenkinek!

Tartalomjegyzék

1. fejezet

Bevezetés

Képzeljük el, hogy üzenetet szeretnénk küldeni egy barátunknak. Ezt egy kommunikációs csatornán keresztül tehetjük meg, ez az a hely, ahol az információcsere létrejöhet. A gyakorlatban előfordul, hogy ez a csatorna zajos, ami annyit jelent, hogy az az információ, amit küldeni szeretnénk, sok esetben megsérül, tehát nem ugyanaz érkezik meg a címzetthez, mint amit a feladó elküldött. Ezeket a rendellenességeket a hibajavító kódok alkalmazásával tudjuk kiküszöbölni.
A hibajavító kódok célja, hogy minél több hiba javítható legyen és ennek ellenére ne hosszabbodjon túlzott mértékben az üzenet, valamint a kódolás illetve dekódolás futási ideje is megfelelő gyorsaságú legyen.
Ebben a dolgozatban ezt a hibajavító képességet vizsgáljuk részletesebben.

2. fejezet

Kódelméleti alapok

Ebben a fejezetben általánosan leírom, hogyan működik az a folyamat, amikor elküldünk egy kódolt üzenetet, amely később megsérül, majd bemutatom, hogyan lehet ezeket a hibákat kijavítani és a dekódolást elvégezni. A közérthető bevezető után, a későbbi fejezetekben a hibajavító kódok egyik legismertebb fajtáját, a Reed-Solomon kódokat ismertetem.

2.1. Kódolás

Most már, hogy tudjuk miért hasznosak a hibajavító kódok, lássuk, hogyan működnek.
A hibajavító kódok az eredeti üzenetet először matematikailag értelmezhető kifejezéssé alakítják át, ezt a fajta kódolást forráskódolásnak nevezzük.

1. Definíció (Forrásábécé)**.** *Forrásábécé*nek nevezzük azoknak a karaktereknek az összességét, amelyekből az eredeti üzenet áll. Jelöljük a forrásábécét X-el, az elemeiből képzett véges sorozatok halmazát pedig X^*-gal. Ez lesz az üzenetek halmaza.

Például ha az üzenetünk a következő: „Milyen szép napos időnk van.”, akkor a forrásábécé, amit használunk a magyar ábécé.

2. Definíció (Kódábécé)**.** *Kódábécé*nek nevezzük azokat a karaktereket, amelyből a kódszó áll. Jelöljük a kódábécét Q-val, az ebből alkotott véges sorozatokat pedig Q^*-gal, ahol $|Q| = q$.

Gyakran bináris karakterekké alakítjuk a szöveges üzenetet, ezért a kódábécénk sok esetben a $\{0, 1\}$ halmaznak felel meg.

3. Definíció (Forráskódolás)**.** *Forráskódolás*nak egy olyan injektív függvényt nevezünk, amely egy üzenethez kódszót rendel. Jelölése: $f : X \to Q^*$.

Itt elég megadni f-et X-en, abból f^*-ra való kiterjesztése konkatenálással kapható. Ezt betűnkénti kódolásnak szokták nevezni.

Megkülönböztetünk változó és egyenlő szóhosszúságú forráskódolásokat. Itt a különbséget, a forrásábécé karaktereihez rendelt számsorozatok hossza jelenti. [3]
Vegyük például a „nap” szót. Legyen $f(n) = 0$, $f(a) = 1$ és $f(p) = 01$. Tehát a „nap” üzenet forráskódolással kódszóvá alakítva a 0101 számsorozat. Ez egy változó szóhosszúságú forráskódolásra példa, melynek előnye, hogy egy gyakrabban használt betűnek a forráskódolással előállított kódszava rövidebb, míg a ritkábban használt karakterek kaphatnak hosszabb számsorozatokat is. Ez jelentős memóriahely megtakarítást jelenthet hosszabb üzenet esetén.
A kódábécét meg tudjuk feleltetni egy véges test elemeinek. Például a $\{0, 1\}$-re gondolhatunk, úgy mint a *mod* 2 testre. Ezen gondolat képezi az alapját a hibajavító kódok alkalmazásának. Ezt a nyolcadik fejezetben részletesen ismertetem.
Mivel az üzenet terjedelme változó, nem csak egy szó, hanem egy mondat vagy akár több oldal is lehet, ezért a számsorozatot fel kell osztanunk apróbb darabokra, blokkokra. Így, azokat külön-külön elkódolva továbbíthatjuk a címzettnek. Ezt a kódolást blokk-kódolásnak nevezzük.

4. Definíció (Blokk-kódolás)**.** Az üzenet, már forráskódolt (bináris) számsorozatait, k méretű blokkokra bontjuk, majd ezeket rendeljük hozzá N hosszú sorozatokhoz, ahol $k < N$. A megfeleltetés itt is egyértelmű. Jelölése: $g : Q^k \to Q^N$.

Itt hiába $k < N$, a blokk-kódolással keletkezett kódszó nem hordoz több információt, hanem a meglévő tartalmat kódolja hosszabban. Ezt *redundáns*nak nevezzük, emiatt fogjuk tudni dekódoláskor visszanyerni az eredeti üzenetet, még ha meg is sérül az zajos csatornán. [6]
Nézzünk erre is egy példát! Az egyik legegyszerűbb példa hibajavító kódokra a három hosszú ismétléses kód. Ez annyit jelent, hogy a blokkokat egy hosszúságúnak választjuk és megháromszorozzuk azt a karaktert. Szóval legyen mondjuk a 01 a forráskódolt üzenet, amelyet blokk-kódolni akarunk, akkor a $g(01) = 000111$. Ezen a kódon szemléletesen látszik, hogy 1 hibát tud javítani, hiszen ha három egyforma szám közül egy megsérül a többség miatt korrigálható a hiba. Továbbá a kód 2 hiba érzékelő, mert három egyforma számból ha kettő megsérül, akkor az látszik a kódból, hogy ott hiba történt, viszont dekódolni helyesen itt már nem tudjuk.

5. Definíció (t-hibajavító)**.** Ha bárhogy veszek egy kódszót és legfeljebb t helyen tetszőlegesen megváltoztatom, majd ez a sérült kódszó még mindig egyértelműen visszavezethető az eredeti kódszóra, akkor a kód t-hibajavító.

6. Definíció (h-hibaérzékelő)**.** Vegyünk egy kódszót és legfeljebb h helyen tetszőlegesen megváltoztatjuk. Ha így nem jön létre egy másik értelmes kódszó, akkor a kódunk h-hibaérzékelő.

2.2. Zajos csatorna

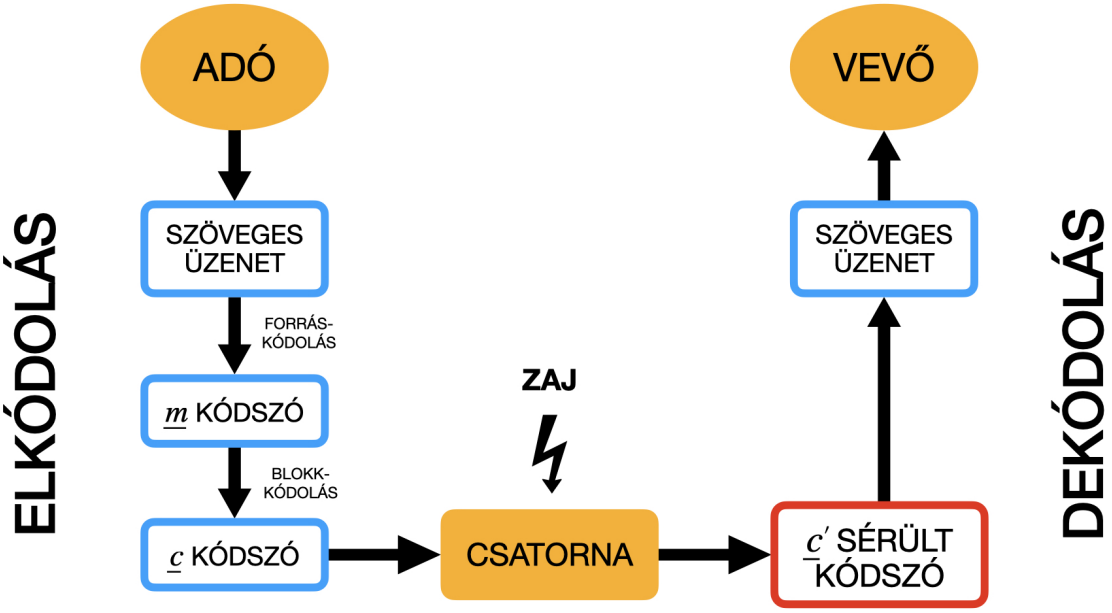
Előfordulhat, hogy az üzenetet nem éri hiba, vagy esetleg kevesebb hiba történik, mint amennyit maximálisan javítani tudna az algoritmus. Ez azonban nem jelent gondot, mivel minden hibahatár alatti sérülésszámot képes javítani a kód és visszaadja az eredeti üzenetet. Ha egy kód maximális hibajavító képessége t, akkor a kódot t-hibajavító kódnak nevezzük.
A csatorna hibáinak jellegéről tudjuk, hogy nem determinisztikusak, tehát véletlenszerűen keletkeznek. Továbbá feltételezzük, hogy hiba esetén karakter nem vész el, csak átalakul. Ezt úgy kell elképzelni, hogy egy bináris számsorozatban a nullák egyesekké, az egyesek pedig nullákká változhatnak.
Létezik azonban olyan eset is, amikor egy hiba nem másik karakter formájában jelentkezik, hanem pusztán olvashatatlanná válik az adott elem. Ezt *törléses hibának* nevezzük, melynek dekódolásakor a hibahely detektálás ránézésre is látszik, ezért csak az algoritmus ezt követő részét kell végrehajtani.
Továbbá ismert hibajelenség még a *hibacsomó*, amelynek jellegzetessége, hogy a sérülések közvetlen egymás után következnek. Erre jó példa, amikor egy hangfelvételben a háttérben becsapódik az ajtó és pár másodpercig nem hallható az eredeti hang.

2.3. Dekódolás

Abban az esetben, amikor az elküldött kód megsérül a csatornán, dekódolás alatt nem elég pusztán arról beszélnünk, hogy a kódolt üzenetből megkapjuk az eredeti tartalmat. Mivel ezt a kódot visszafejtve nem az eredeti üzenetet kapjuk vissza, ezért először a hibajavítást kell elvégeznünk. Tehát e két folyamat együttese adja egy hibajavító kód dekódolását.
A hibajavítás algoritmusa két alapvető komponensből áll. Kezdetben a hibának a helyét határozzuk meg, hogy mely karakterek vagy karaktersorozatok sérülhettek. Ebből az információból tehát az is kinyerhető, hogy melyek azok az elemek, amelyek sértetlenek. Ezt követően fejtjük meg azt, hogy ezeken a helyeken mi állhatott eredetileg. A két adatot összegyúrva kapjuk meg a kódszót, amit még a feladó küldött el a zajos csatornán. Innentől kezdve már egyszerű a dolgunk, hiszen visszafejtjük a kódolt üzenetet és már el is olvashatjuk, hogy mi a nekünk küldött információ.

2.4. Áttekintés

A könnyebb megértés érdekében az eddig tárgyalt lépéseket az alábbi ábra szemlélteti:



2.1. ábra. Hibajavító kód útja az adótól a vevőig

3. fejezet

Véges testek algebrája

3.1. Alafogalmak

Az algebrában egy *test* olyan $(K, +, \cdot)$ struktúrát jelöl, amelyben két műveletet értelmezünk: az összeadást $(+)$ és a szorzást $(\cdot)$. A $(K, +)$ egy kommutatív csoportot alkot, ennek neve legyen additív csoport. Továbbá a $(K\backslash\{0\}, \cdot)$ egy kommutatív csoportot alkot, ennek neve legyen multiplikatív csoport. Ha a multiplikatív csoport a kommutatív tulajdonságot nem teljesíti, akkor *ferdetest*ről beszélünk. Ezen felül az összeadás és a szorzás műveletére teljesül egy további tulajdonság, a disztributivitás, miszerint: $a \cdot (b + c) = a \cdot b + a \cdot c$. Ha egy testnek véges sok eleme van *véges test*nek nevezzük.

3.1.0.1. Állítás (Véges testek alaptétele)**.** *Legyen K egy véges test. Ekkor létezik p prím, melyre igaz, hogy $|K| = p^n$, ahol $n \in \mathbb{N}^+$. Másrészt minden $q = p^n$-hez létezik pontosan egy olyan K véges test, melyre igaz, hogy $|K| = q$.*
Jelölés: $GF(q)$, $\mathbb{F}_q$, elnevezése Galois test.

3.2. Példák

Nézzünk rájuk néhány példát!
Elsőként tegyük fel, hogy p prímszám, ekkor a $(mod\ p, +, \cdot)$ test, tehát a $mod\ p$ maradékosztályok testet alkotnak.
Második példánk legyen a $K = GF(4)$, ahol $f(x) = x^2 + x + 1$ irreducibilis $GF(2)$ fölött. Elemei $0, 1, x, x + 1$. Ekkor a K additív és multiplikatív csoportjai a következők:

Additív csoport:

+	0	1	x	x+1
0	0	1	x	x+1
1	1	0	x+1	x
x	x	x+1	0	1
x+1	x+1	x	1	0

Multiplikatív csoport:

·	1	x	x+1
1	1	x	x+1
x	x	x+1	1
x+1	x+1	1	x

Ennek a témakörnek a részletesebb leírását megtalálják a Kiss Emil *Bevezetés az algebrába* című könyvében.[2]
A fenti, GF(4) előállítására használt eljárást általánosítani lehet $q = p^h$, p prím, elemű testek előállítására. Legyen $f(x)$ irreducibilis $GF(p)$ fölött, $deg(f) = h$. Ekkor a $GF(q)$ test a $GF(p)[x]/(f(x))$ faktorgyűrű lesz. Elemei tehát a h-nál kisebb fokú polinomok, amelyekkel modulo $f(x)$ számolunk, a polinomok együtthatóival pedig persze modulo p. Ez azt jelenti, hogy ha pl. $a(x)b(x)$ h-nál nagyobb fokú, akkor a szorzás eredménye az $a(x)b(x)$ szorzat $f(x)$-szel való osztásának maradéka lesz. Ezt illusztrálja a fenti példa is, ott pl. $x(x+1) = 1(x^2+x+1)+1$, vagyis a maradék 1, így az $x(x+1)$ szorzat is 1 lesz. Ezt az elvet követjük más műveleteknél is, ha az eredmény polinom foka h vagy annál nagyobb, akkor helyettesítjük az f(x)-szel való osztás maradékával. Jegyezzük meg, hogy ugyanezt csináljuk a 8.2. szakaszban is, amikor $64 = 2^6$ elemű testet állítottunk elő, az $f(x) = x^6 + x + 1$ irreducibilis polinom felhasználásával. Ebben a testben a nem-nulla elemeket fel tudjuk írni, mint az x polinom (az ottani táblázatban a g elem) valamelyik hatványát. Az imént említett gondolatmenet Szőnyi Tamás jegyzete alapján készült.[5]

4. fejezet

Lineáris kódok

4.1. Véges testek fölötti vektorterek

Legyen $V = V(n, q)$ egy n dimenziós vektortér a $GF(q)$ fölött. Ennek elemei n hosszú sorvektorok, ahol $|V| = q^n$. Legyen U a $\underline{g}_1, \underline{g}_2, \ldots, \underline{g}_k$ lineáris független vektorok által generált k dimenziós altere a V-nek.

7. Definíció (Vektorok skaláris szorzata)**.** Legyen $\underline{a} = [a_1, a_2, \ldots, a_n]$ és $\underline{b} = [b_1, b_2, \ldots, b_n]$ két n dimenziós valós vektor. Ezen vektorok skaláris szorzatán az $\underline{a} \cdot \underline{b}^T = a_1 \cdot b_1 + a_2 \cdot b_2 + \ldots + a_n \cdot b_n$ mennyiséget értjük.

8. Definíció (Mátrix szorzása mátrixal)**.** Legyen $A \in \mathbb{R}^{n \times k}$ és $B \in \mathbb{R}^{k \times l}$ mátrixok. Legyen $C \in \mathbb{R}^{n \times l}$ ezen mátrixok szorzata, amely mátrix i-edik sorának j-edik elemét úgy kapjuk meg, hogy az A mátrix i-edik sorát „skalárisan szorozzuk" a B mátrix j-edik oszlopával.[7]

Ekkor $G = \begin{pmatrix} \underline{g}_1 \\ \ldots \\ \underline{g}_k \end{pmatrix}$ $k \times n$-es mátrix az altér egy generátormátrixa. Ez annyit jelent, hogy ha van egy $\underline{x} \in Q^k$ k hosszú vektorunk, akkor $x \cdot G \in Q^n$ n hosszú kódszót generál belőle. Ha $\underline{x} = (x_1, x_2, \ldots, x_k)$, akkor épp az $x_1 \cdot \underline{g}_1 + x_2 \cdot \underline{g}_2 + \ldots + x_k \cdot \underline{g}_k$-t.

9. Definíció (Kódszavak közti minimális távolság)**.** Jelölje d a *kódszavak közti minimális távolság*-ot. Ekkor $d = \min\limits_{\substack{\underline{u}, \underline{u}' \in C \\ \underline{u} \neq \underline{u}'}} d(\underline{u}, \underline{u}')$

10. Definíció (Kód duálisa)**.** A C kód *duálisán* azt a $C^\perp = \{v \in Q^N : v \cdot c = 0$ minden $c \in C$ kódszóra $\}$ kódot értjük, mely egy lineáris kód. A $C^\perp$ kód H generátormátrixát a C kód ellenőrző mátrixának nevezzük. [4]

Legyen $U^{\perp} = \langle \underline{h}_1, \underline{h}_2, \ldots, \underline{h}_{n-k} \rangle$, ekkor $H = \begin{pmatrix} \underline{h}_1 \\ \ldots \\ \underline{h}_{n-k} \end{pmatrix}$ egy $(n-k) \times n$-es mátrix a paritás ellenőrző mátrix. Ez a mátrix pedig leellenőrzi egy adott vektorról, hogy kódszó-e, vagyis $x \in C$ akkor és csak akkor, ha $x \cdot H^T = 0$. Ugyanazon altér generátor és paritásellenőrző mátrixa esetén $G \cdot H^T = 0$ ($k \times (n-k)$-as csupa nulla mátrix). Továbbá G-ből megalkotható H a következő képpen:

$$G = \left(\begin{array}{c|c} I_k & A \end{array} \right) \rightarrow H = \left(\begin{array}{c|c} -A^T & I_{n-k} \end{array} \right),$$

melyekben az identitás mátrix bárhol helyezkedhet el, nem feltétlen szerepel az elején vagy a legvégén. Mindkét mátrixban a sorok függetlenek. Az alábbi tárgyalás Szőnyi Tamás jegyzete alapján készült. [5]

4.2. Lineáris kódok

11. Definíció (Lineáris kód)**.** Legyen $Q = GF(q)$. Ekkor Q^n egy $GF(q)$ test feletti vektortér. C *lineáris kód*, ha C a Q^n k dimenziós altere.
Jelölése: $[n,k]_q$ vagy $[n,k,d]_q$, ha a minimális távolságot is feltüntetjük.

12. Definíció (Kód minimális súlya)**.** Egy *kód minimális súlya* a nem-nulla kódszavak súlyának minimuma. Lineáris kódokra ez megegyezik a kód minimális távolságával.

A Reed-Solomon kódok lineáris kódok, amely nagy segítséget nyújt a generálásához, ellenőrzéséhez valamint az algoritmus futása alatti számolásokhoz.
Vezessünk be további két fogalmat: a Singleton-korlátot, valamint az MDS tulajdonságot.

13. Definíció (Singleton-korlát)**.** Ha C egy $[n,k,d]_q$ lineáris kód, akkor a paritásellenőrző mátrixban legfeljebb $(n-k)$ oszlop lehet független, amiből következik, hogy $d \leq n-k+1$.

14. Definíció (Max Distance Separable)**.** Ha C egy $[n,k,d]_q$ lineáris kód és a paritásellenőrző mátrixának bármely $(n-k)$ oszlopa független, akkor teljesül rá az MDS tulajdonság, ami azt jelenti, hogy $d = n-k+1$. Ez megfordítva is így van $d = n-k+1$ esetén a paritásellenőrző mátrix bármely $n-k$ oszlopa független.

Az MDS tulajdonság teljesülésekor a paritásellenőrző mátrixba bárhová betranszformálható az identitás mátrix. Ekkor a duális kód paritásellenőrző mátrixába is mindenhova betranszformálható az identitás mátrix. Ebből az következik, hogy egy MDS kód duálisa is MDS kód. [8]

5. fejezet

Kódolás

A kódelmélet világában rentgeteg algoritmus áll a rendelkezésünkre, amellyek nem csak az üzeneteink nyilvánosság elől való elrejtését teszik lehetővé, hanem képesek a sérülések esetén történő hibajavításra is. Mint már korábban említettem, ezek közül is a Reed-Solomon kódokat mutatom be. Ehhez elsőként értsük meg a kód felépítését.

5.1. Reed-Solomon kódok felépítése

A Reed-Solomon hibajavító kódok egy csoportját Irving S. Reed és Gustave Solomon mutatták be 1960-ban. Ezen algoritmusok számtalan alkalmazási területnek lettek kulcsfontosságú eszközei. Többek között a CD-k, DVD-k, valamint vonalkódok és QR kódok is használják azt a képességét, hogy még sérült adathordozókból is kinyerhető az információ. Erről bővebben a Hraskó András által szerkeztett *Új matematikai mozaik* című könyvben olvashatnak. [9]

Legyen (N, k) t-hibajavító Reed-Solomon kód, ahol minden műveletet egy $GF(q)$ véges test fölött végzünk. Ebben a kifejezésben k jelölje az elküldeni kívánt üzenet, N pedig a kódszó hosszát.

$$\underline{m} = (m_1, m_2, \ldots, m_k) \in GF(q)^k$$
$$\underline{c} = (c_1, c_2, \ldots, c_N) \in GF(q)^N$$

Fontos, hogy a t-hibajavító képesség a kódszóban és nem az üzenetben történő hibákat számszerűsíti. A véges testet válasszuk aszerint, hogy az elemszáma legyen nagyobb vagy egyenlő az ábécénk méretével. Itt érdemes kettőhatvány méretű ábécével dolgozni, hiszen így optimalizálhatjuk a kifejezhető karakterek számát, a használt véges test elemszámával. Ezek után feleltessük meg az ábécé karaktereit a test elemeinek, azaz a forráskódolással már a karakterekből testelemeket képzünk.

Most lássuk, hogyan működik a blokk-kódolás.
Definiáljunk két újabb kifejezést, az $f_m(x)$-et, amelyben az üzenetet egy $\leq k-1$ fokú polinom együtthatóiként tekinthetjük és $f_m : GF(q) \mapsto GF(q)$, valamint az $\underline{\alpha}$ vektort.

$$f_m(x) \coloneqq m_0 + m_1 x + \ldots + m_{k-1} x^{k-1}$$
$$\underline{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_N) \in GF(q)^N, \alpha_i \neq \alpha_j, N \leq q$$

Az $\underline{\alpha}$ elemeit a véges testből választjuk, ezért érdemes ügyelni a megválasztására. Szokás őket N-edik egységgyökökként definiálni, mert ezek eloszlása lesz a legegyenletesebb az ábécén. Nevezzük el c_i-nek az $f_m(x)$ függvény értékét az $\alpha_1, \alpha_2, \ldots, \alpha_N$ helyeken:

$$c_1 = f_m(\alpha_1)$$
$$c_2 = f_m(\alpha_2)$$
$$\vdots$$
$$c_N = f_m(\alpha_N)$$

A fenti megfeleltetéssel a $\underline{c} = (c_1, c_2, \ldots, c_N)$ kódszó összes koordinátáját sikeresen meghatároztuk, így készen vagyunk a kódolással. A kapott kód lineáris, k-dimenziós, N hosszú, azaz $[N, k]_q$ kód.

5.1.0.1. Állítás. *A Reed-Solomon kódok MDS kódok.*

Bizonyítás: Lehet-e egy fenti módon kapott nem-nulla kódszó súlya kevesebb, mint $n - k + 1$, azaz legfeljebb $n - k$? Ez azt jelentené, hogy az $f_m(x)$ polinom legalább k helyen nulla (az α-k közül). Mivel $f_m(x)$ foka $\leq k - 1$, ekkor $f_m(x)$ a nulla polinom lenne, azaz a kapott kódszó is a csupa nulla lenne.

6. fejezet

Csatorna

A kódszavakon ejtett hibák precízebb megértése érdekében egy szemléletes modellt fogok bemutatni, ami a kód hátterében futó algebrai világot reprezentálja.

Képzeljük el az ábécéből kifejezhető, adott N hosszúságú szavak összességét, mint egy nagy halmazt. Ez a $GF(q)^N$, amit a továbbiakban jelöljünk csak Q^N-el. Ebben pontokat alkotnak a szavak és a kódszavak. Az utóbbinak a halmazát jelöljük C-vel. Ezt a Q^N halmazt csoportosítsuk részhalmazokra úgy, hogy minden kódszóhoz tartozzon pontosan egy részhalmaz. Ezekben a további elemek aszerint rendeződjenek, hogy a kódszó és bármely másik szó, ami benne van a részhalmazban, legfeljebb t darab karakterben különbözzön egymástól. Ezt a fajta eltérést Hamming-távolságnak, a részhalmazokat, amiket képeztünk, pedig Hamming-gömbnek nevezzük. Ha ezek a gömbök diszjunktak, akkor az így létrehozott kód t-hibajavító.

15. Definíció (Hamming-távolság)**.** Legyen $\underline{x} = (x_1, x_2, \ldots, x_n)$ és $\underline{y} = (y_1, y_2, \ldots, y_n)$, ahol $\underline{x}, \underline{y} \in Q^N$. Ekkor a $d(\underline{x}, \underline{y}) = |\{i : x_i \neq y_i\}|$ kifejezést *Hamming-távolságnak* nevezzük.
Tulajdonságai:

- $d(\underline{x}, \underline{y}) \geq 0$
- $d(\underline{x}, \underline{y}) = 0 \Leftrightarrow \underline{x} = \underline{y}$
- $d(\underline{x}, \underline{y}) = d(\underline{y}, \underline{x})$
- $d(\underline{x}, \underline{y}) \leq d(\underline{x}, \underline{z}) + d(\underline{z}, \underline{y})$ (háromszög-egyenlőtlenség)

Az imént említett definícióban leírt tulajdonságok azt mutatják, hogy a Hamming-távolság bizonyos tekintetben úgy viselkedik, mint a szokásos euklideszi távolság.
Mostanra nagyobb algebrai rálátással bírunk, tehát definiáljuk a t-hibajavító képességet precízebben:

16. Definíció (t-hibajavító kód)**.** Ha minden olyan $\underline{x} \in Q^N$-re, amelyhez létezik $\underline{c} \in C$ kódszó, amelyre $d(\underline{x}, \underline{c}) \leq t$, a $\underline{c} \in C$ egyértelmű (azaz $\underline{c}' \neq \underline{c}$-re $d(\underline{x}, \underline{c}') > t$), akkor C-t t-hibajavító kódnak nevezzük.

Tehát ha a kódszavunk legfeljebb t helyen sérül, akkor az egyértelmű leképezés, ami kódszót üzenetbe képez (dekódolás), a helyes üzenetet fogja visszaadni. Ebből látszik, hogy t hibáig megfelelően javít, tehát t-hibajavító.
A Hamming-távolság definícióját felhasználva definiáljuk a kódszavak közti minimális távolságot:

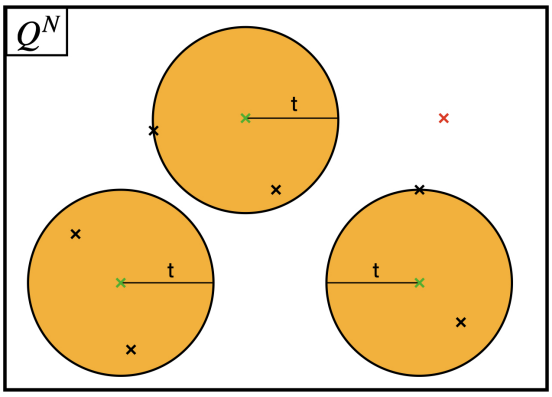
17. Definíció (Kódszavak közti minimális távolság)**.** Jelölje d a *kódszavak közti minimális távolság*-ot. Ekkor $d = \min\limits_{\substack{\underline{u},\underline{u}' \in C \\ \underline{u} \neq \underline{u}'}} d(\underline{u}, \underline{u}')$.

A t-hibajavító képesség és a kódszavak közti minimális távolság összefüggéséről az alábbi állítás szól:

6.0.0.1. Állítás. *A kódszavak közti minimális távolság nagyobb vagy egyenlő, mint a kód hibajavító képességének a kétszerese plusz egy, azaz* $t = \left\lfloor \frac{d-1}{2} \right\rfloor \Leftrightarrow d \geq 2t + 1$.

18. Definíció (Hamming-gömb)**.** Legyen $\underline{c} \in C$ kódszó, $\underline{x} \in Q^N$ szó, valamint legyen $t > 0$ egész szám. Ekkor a $B(\underline{c}, t) = \{\underline{x} : d(\underline{c}, \underline{x}) \leq t\}$ kifejezést *$\underline{c}$ középpontú, t sugarú Hamming-gömb*-nek nevezzük. Itt $|B(\underline{c}, t)| = \sum\limits_{i=0}^{t} \binom{N}{i} (q-1)^i$.

Tehát az $\underline{x}$ vektor koordinátái közül kiválasztunk i darabot, amelyek mindegyikét $(q-1)$-félére cserélhetünk, így kapjuk meg a Hamming-gömb elemszámát.



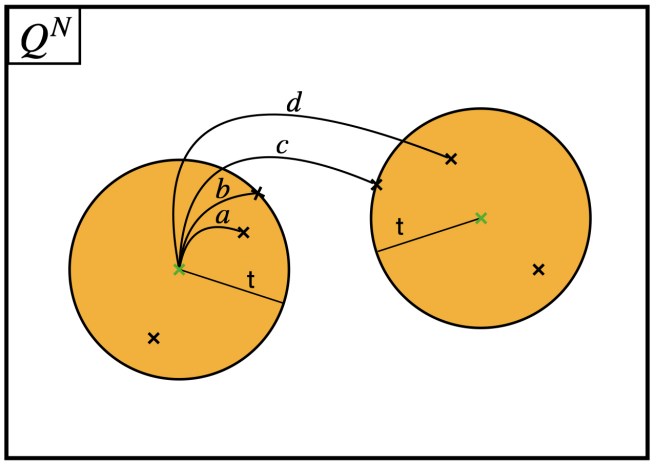
6.1. ábra. Az N hosszú szavak alaphalmazában a narancssárga részhalmazok a Hamming-gömböket szimbolizálják, közepükön a zöld keresztek az adott halmazhoz tartozó kódszót jelölik, diszjunkció esetén a piros kereszttel jelzett pont nem létezik

Tehát a Hamming-gömbökre nézve a t-hibajavító tulajdonság pontosan azt jelenti, hogy a kódszavak köré írt t sugarú gömbök (páronként) diszjunktak.

19. Definíció (Hamming-korlát)**.** Ha $C \subseteq Q^N$ t-hibajavító kód, akkor igaz rá, hogy $|C| \cdot |B(\underline{c}, t)| \leq |Q|^N$. Ezt az egyenlőtlenséget *Hamming-korlát*nak nevezzük.

20. Definíció (Perfekt-kód)**.** Ha $C \subseteq Q^N$ t-hibajavító kód és igaz rá, hogy $|C| \cdot |B(\underline{c}, t)| = |Q|^N$, akkor a kód *perfekt*.

A korábban említett három hosszú ismétléses kód jó példa a perfekt tulajdonságra, mivel 1-hibajavító és $|C| = 2$, $|B(\underline{c}, t)| = 4$, valamint $|Q|^N = 2^3$, tehát teljesíti az egyenlőséget. Amikor egy ilyen kódszót kívánunk elküldeni a csatornán, majd az megsérül, akkor annyi történik, hogy elcsúszik az üzenet és attól függően, hogy hány helyen sérül, egy másik pontra ugrik a halmazunkban. Tegyük fel, hogy nem kap t-nél több hibát, ekkor benne marad a Hamming-gömbben, tehát egyértelműen visszanyerhető belőle az eredeti kódszó, hiszen az elején még kikötöttük, hogy minden kódszóhoz szigorúan egy részhalmaz tartozzon. Ez az eset a fejezet végén található ábrán a és b betűkkel van ellátva. Azonban ha t-nél több hiba keletkezik a kódszóban, akkor már átugrunk egy másik Hamming-gömbbe ahonnan már nem visszafejthető az üzenet. Ez az eset az ábrán c és d betűkkel látható. Tehát ez jelenti azt, hogy a kódunk t-hibajavító. Habár persze mindig megvan az a hajszálnyi esély arra, hogy az üzenetünk pont úgy sérül, hogy egy hibát a következő sérülés visszakorrigál, tehát kettő hiba helyett egy sem történik. Ilyenkor persze semmi teendőnk vele hiszen ugyanaz a tartalom fog megérkezni a vevőhoz, amit a feladó elküldött.



6.2. ábra. Két Hamming-gömb hibajavítás során látható, a és b eset: t vagy annál kevesebb hiba történt, még dekódolható az üzenet, c és d eset: t-nél több hiba történt, az üzenet már nem visszafejthető

7. fejezet

Dekódolás Reed-Solomon kódokra

Ebben a fejezetben képzeljük el, hogy mi vagyunk a fogadó fél, akihez a kódszó érkezik, azonban egy olyan üzenetet kapunk, amiből hibajavítás nélkül nem nyerhető ki értelmes tartalom. Tehát lássuk, hogyan dekódoljunk egy megsérült kódszót.

Először is mi az, amit ismerünk? Nevezzük $\underline{c}'$-nek azt a szót, ami az eredeti kódszó megsérült változataként érkezik hozzánk. Ennek tudjuk a paramétereit, tehát az N és k is ismert. Továbbá azt tudjuk, hogy t vagy annál kevesebb hiba történhet a csatornán, ellenben azt hogy ezek hol vannak, vagyis mely karakterek sérültek, azok ismeretlenek számunkra. Ezen felül még egy ismert paraméter áll a rendelkezésünkre, ami az $\underline{\alpha}$ vektor.
Mielőtt nekikezdünk a dekódolásnak fontos ellenőrizni egy korlátot, ami ha nem teljesül a kódunkra, akkor nem tudjuk elvégezni a hibajavítást. A korlát az alábbi:

$$N - k \geq 2t.$$

A fenti egyenlőtlenség miatt, az MDS tulajdonság ($d = N - k + 1$) a következőre írható át: $d \geq 2t + 1$, amely éppen a 6.0.0.1. Állítás-sal egyezik meg ($t = \left\lfloor \frac{d-1}{2} \right\rfloor$).
Tehát a fent kiemelt egyenlőtlenség annyit jelent, hogy a blokk-kódolás során a kódszó és az üzenet hosszának a különbsége, legalább kétszer akkora legyen, mint a hibák száma. Ha ez teljesül, akkor nekikezdhetünk a dekódolásnak.
Definiáljunk egy pár függvényt a szemléletesség kedvéért! Legyen $f_m(x)$ továbbra is az üzenetet elkódoló polinom, amit korábban is használtunk, foka legfeljebb $k-1$. Legyen $R(x)$ az az $N-1$ fokú polinom, ami $\underline{\alpha}$-ból a $\underline{c}'$-be képez, az alábbi módon:

$$\begin{aligned} c'_1 &= R(\alpha_1), \\ c'_2 &= R(\alpha_2), \\ &\vdots \\ c'_N &= R(\alpha_N). \end{aligned}$$

Nem több, mint t hely kivételével $\underline{c}$ megegyezik a $\underline{c}'$ vektorral.

Ezen felül legyen $E(x)$ egy t-ed fokú hibahely függvény, amely adott α_i-hez akkor rendel nulla testelemet, ha ott hiba történt és nem nulla testelemet, ha a kódszó ott sértetlen.

$$E(x) = (x - e_1)(x - e_2)\dots(x - e_t),$$

ahol $e_1 = \alpha_{i_1}, e_2 = \alpha_{i_2}, \dots, e_t = \alpha_{i_t}$ alkalmas $i_1, i_2, \dots, i_t$-re.
Ha t-nél kevesebb hiba történt, akkor is pótoljuk az ismeretleneket, míg a fent látható alakú nem lesz a hibahely függvény, azaz ha $E(\alpha_i) = 0$, akkor nem biztos, hogy ott tényleg hiba történt. A fenti függvények közül, számunkra csak az $R(x)$ ismert. Most lássuk, hogyan kapjuk meg a polinomok és a sérült kódszó segítségével az eredeti kódszót.

$$f_m(x) \cdot E(x) = R(x) \cdot E(x),$$

ahol $x = \alpha_i, i = 1, 2, \dots, N$.
Itt olyan α_i-re, ahol nem történt hiba $f_m(x)$ megegyezik $R(x)$-el, ahol pedig hiba történt az $E(x)$ nulla értéket vesz fel, ezáltal azonosan nullával szintén egyenlő a két oldal. Nevezzük ezentúl az $f_m(x) \cdot E(x)$ szorzatot $Q(x)$-nek. Fejtsük ki ezt az egyenletrendszert!

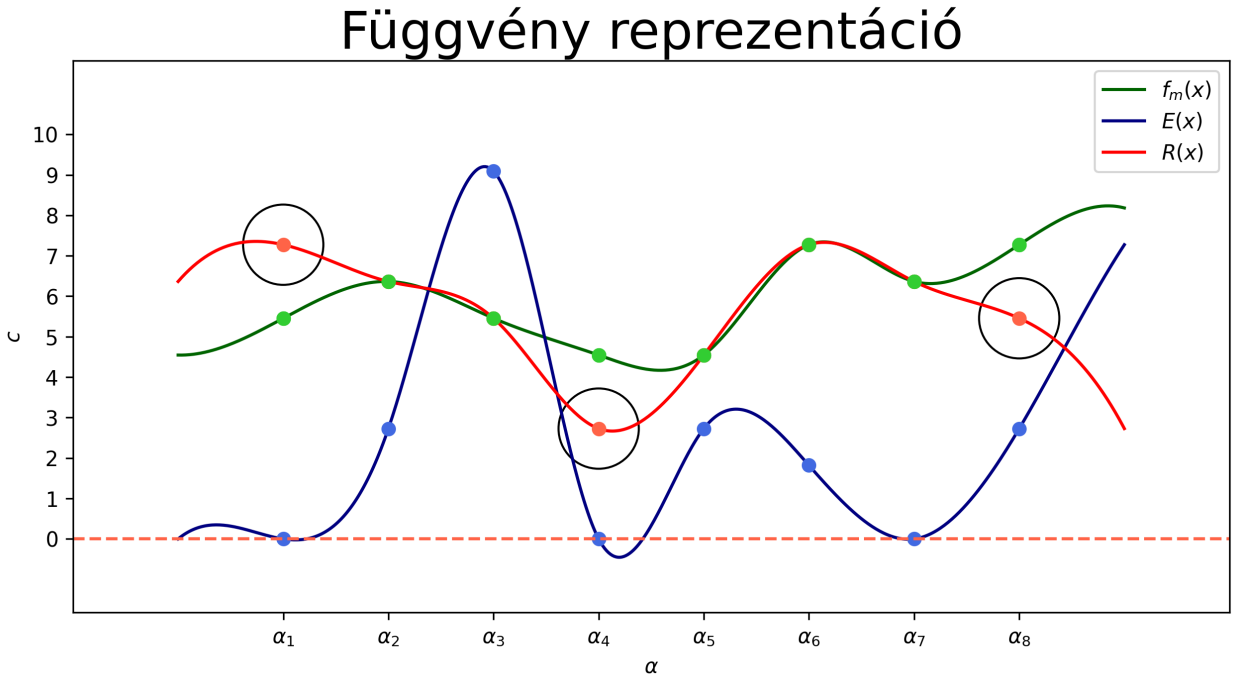
$$\begin{aligned} Q(\alpha_1) &= R(\alpha_1) \cdot E(\alpha_1), \\ Q(\alpha_2) &= R(\alpha_2) \cdot E(\alpha_2), \\ &\vdots \\ Q(\alpha_N) &= R(\alpha_N) \cdot E(\alpha_N). \end{aligned}$$

Ez N darab egyenlet. Valamint, mivel az ismeretlen együtthatók száma $f_m(x)$-ben k darab, $E(x)$-ben t darab, ezért $Q(x)$-ben $k + t$ darab, és így végül a teljes egyenletrendszerben az ismeretlenek száma $k+t+t$. Mivel a fenti kiemelt egyenlőtlenség így szólt, hogy egy kód csak akkor dekódolható, ha $N - k \geq 2t$, ezt átrendezve azt kapjuk, hogy $N \geq k + t + t$, ezért a kapott egyenletrendszer a szükséges feltételt teljesíti. Azért tudjuk, hogy az egyenletrendszer megoldható, mert feltételeztük, hogy $\leq t$ hiba történt, azaz van olyan $\underline{c}$ kódszó (vagyis $f_m(x)$) és hibahely-polinom ($E(x)$), amire fennáll az egyenlőtlenség. Ezen a ponton az is jól látható, hogy ha t-nél kevesebb hiba történt, akkor ez nem jelenthet problémát az egyenletrendszer megoldásánál, hiszen a fent említett egyenlőtlenségben a t változó a jobb oldalán található, ezt csökkentve megmarad az állítás igazságtartalma. Valamint továbbra is feltételezzük, hogy létezik olyn kódszó és hibahely-polinom, amelyre fennáll az egyenlőtlenség.
Ha kiszámoltuk az ismeretleneket, akkor az e_i-k értékei megadják a keletkezett hibák helyét. Most már tudjuk, hogy mik azok a konkrét értékek, amik megegyeznek $\underline{c}$ és $\underline{c}'$-ben, ezen felül azt is tudjuk, hogy melyek azok a $\underline{c}'$ értékek, amiket kihagyva és a sértetlenekkel tovább dolgozva megkaphatjuk az eredeti üzenetet.
Ehhez az $f_m(x)$-et, tehát az üzenet dekódoló polinomot kell megtudnunk, hogy mik lehettek az eredeti együtthatói, hiszen ez éppen az üzenet karaktereit tartalmazza. Készítsünk még

egy egyenletrendszert, az $f_m(x)$ felhasználásaval, úgy hogy ezúttal töröljük ki az $\underline{\alpha}$-k közül azokat, amely helyeken sérült az üzenet, valamint a kódszóból a hibás karaktereket. Így a következőket kapjuk:

$$\begin{aligned} m_0 + m_1\beta_1 + m_2\beta_1^2 + \ldots + m_{k-1}\beta_1^{k-1} &= c_1, \\ m_0 + m_1\beta_2 + m_2\beta_2^2 + \ldots + m_{k-1}\beta_2^{k-1} &= c_2, \\ &\vdots \\ m_0 + m_1\beta_m + m_2\beta_m^2 + \ldots + m_{k-1}\beta_m^{k-1} &= c_m, \end{aligned}$$

ahol $\beta_i \in \underline{\alpha}$ megfelelelő α-kra. Itt $N-t = m$ az egyenletek száma, az ismeretlen együtthatóból pedig k darab van, ezért az $N-k \geq 2t$-s szabály miatt, az is igaz, hogy $N-k \geq t$, és nekünk éppen ez kellet, vagyis hogy az $m \geq k$ legyen, tehát az egyenletrendszerünk megoldható. Ennek a kiszámítása után megoldásként megkapjuk az $\underline{m}$ vektor elemeit sorra, tehát az eredeti üzenetet.



7.1. ábra. Függvények reprezentációja, ahol az $f_m(x)$ zöld, az $E(x)$ kék és az $R(x)$ pedig piros színnel látható, a feketével bekarikázott pontok pedig az eltérést mutatják, ahol $f_m(x)$ nem egyezik meg $R(x)$-el, tehát ezeken az α helyeken sérült az üzenet (jelen esetben α_1, α_4 és α_8)

8. fejezet

Megvalósítás

A közetkező néhány oldalon a Reed-Solomon hibajavító kódok egy konkrét alkalmazását mutatom be. A Python programnyelven megírt kódom részletesen végig megy az algoritmuson és alkalmazza annak lépéseit. A kódom implementációjában nagy segítségemre volt Ashish Choudhury az Indian Institute of Science professzora, akinek oktatóvideójának köszönhetően elmélyülhettem a Reed-Solomon hibajavító kódok részleteiben. [1] Lássuk hogyan alkalmazzuk a kódot!

8.1. Bemenet

Tegyük fel, hogy mi az alábbi szövegrészletet szeretnénk elküldeni:

The cat is sleeping. The dog is eating.

Azért használok a példában angol nyelvű üzenetet, mert az angol ábécé karaktereit ismeri a kódom. Ezt a későbbiekben egy több elemű véges testet választva és ékezetes karakterekkel bővítve az ábécét, már bármilyen magyar szöveg küldésére is alkalmas lesz. Ahhoz viszont, hogy egy ilyen nagyobb terjedelmű üzenetet, úgy tudjuk kódolni, hogy a kód hibajavító képessége és az algoritmus futásideje is ideális maradjon, fel kell osztanunk a szöveget kisebb egységekre. A választásom a négy karakter hosszú blokkokra esett. Ennek a terjedelemnek az elkódolása nem eredményez túl nagy méretű kódszót, tehát könnyen és gyorsan tudjuk dekódolni még akkor is, ha jelentős százaléka sérül az üzenetnek. Sok esetben az elküldeni kívánt szöveg karakter száma nem lesz osztható néggyel, ilyenkor a megoldás, hogy kipótoljuk szóközökkel az utolsó blokkot, míg négy hosszú nem lesz. Ezt azért tehetjük meg, mert az algoritmusunk a szóközöket is karakterként kezeli.
Ekkor az első mondatot véve a négy karakterenként felbontott üzenet az alábbi lesz:
$[[T, h, e, \textit{szóköz}], [c, a, t, \textit{szóköz}], [i, s, \textit{szóköz}, s], [l, e, e, p], [i, n, g, .]]$ Tehát ha megvannak a négy hosszú tömbök, akkor egyesével kódoljuk és elküldjük őket a csatornán, majd a vevő ezeket dekódolja és konkatenálja a szövegrészleteket, míg meg nem kapja a teljes üzenetet.

8.2. Alkalmazott véges test bemutatása

A kód bázisául szolgáló véges testnek a $GF(2^6)$-t választottam, $f(x) = x^6 + x + 1$ irreducibilis polinommal. Ez több okból is megfelelő választásnak bizonyult. Az első feltételem az volt, hogy kettőhatvány elemszámú testet találjak, hiszen bináris karakterekkel számolunk, továbbá a memória is optimálisabban tud kettőhatvánnyal dolgozni. Végül pedig azok a karakterek, amellyeket hasznosnak találtam belevinni az ábécébe (például angol ábécé kis- és nagybetűi, mondatvégi és mondatközi írásjelek, valamint a szóközt) éppen lefoglalta mind a 64 elemét a testnek. A 64 elemű test pontos reprezentációját az alábbi:

0

$g = x$	$g^{22} = x^5 + x^4 + x^2 + 1$	$g^{43} = x^5 + x^4 + x^2 + x + 1$
$g^2 = x^2$	$g^{23} = x^5 + x^3 + 1$	$g^{44} = x^5 + x^3 + x^2 + 1$
$g^3 = x^3$	$g^{24} = x^4 + 1$	$g^{45} = x^4 + x^3 + 1$
$g^4 = x^4$	$g^{25} = x^5 + x$	$g^{46} = x^5 + x^4 + x$
$g^5 = x^5$	$g^{26} = x^2 + x + 1$	$g^{47} = x^5 + x^2 + x + 1$
$g^6 = x + 1$	$g^{27} = x^3 + x^2 + x$	$g^{48} = x^3 + x^2 + 1$
$g^7 = x^2 + x$	$g^{28} = x^4 + x^3 + x^2$	$g^{49} = x^4 + x^3 + x$
$g^8 = x^3 + x^2$	$g^{29} = x^5 + x^4 + x^3$	$g^{50} = x^5 + x^4 + x^2$
$g^9 = x^4 + x^3$	$g^{30} = x^5 + x^4 + x + 1$	$g^{51} = x^5 + x^3 + x + 1$
$g^{10} = x^5 + x^4$	$g^{31} = x^5 + x^2 + 1$	$g^{52} = x^4 + x^2 + 1$
$g^{11} = x^5 + x + 1$	$g^{32} = x^3 + 1$	$g^{53} = x^5 + x^3 + x$
$g^{12} = x^2 + 1$	$g^{33} = x^4 + x$	$g^{54} = x^4 + x^2 + x + 1$
$g^{13} = x^3 + x$	$g^{34} = x^5 + x^2$	$g^{55} = x^5 + x^3 + x^2 + x$
$g^{14} = x^4 + x^2$	$g^{35} = x^3 + x + 1$	$g^{56} = x^4 + x^3 + x^2 + x + 1$
$g^{15} = x^5 + x^3$	$g^{36} = x^4 + x^2 + x$	$g^{57} = x^5 + x^4 + x^3 + x^2 + x$
$g^{16} = x^4 + x + 1$	$g^{37} = x^5 + x^3 + x^2$	$g^{58} = x^5 + x^4 + x^3 + x^2 + x + 1$
$g^{17} = x^5 + x^2 + x$	$g^{38} = x^4 + x^3 + x + 1$	$g^{59} = x^5 + x^4 + x^3 + x^2 + 1$
$g^{18} = x^3 + x^2 + x + 1$	$g^{39} = x^5 + x^4 + x^2 + x$	$g^{60} = x^5 + x^4 + x^3 + 1$
$g^{19} = x^4 + x^3 + x^2 + x$	$g^{40} = x^5 + x^3 + x^2 + x + 1$	$g^{61} = x^5 + x^4 + 1$
$g^{20} = x^5 + x^4 + x^3 + x^2$	$g^{41} = x^4 + x^3 + x^2 + 1$	$g^{62} = x^5 + 1$
$g^{21} = x^5 + x^4 + x^3 + x + 1$	$g^{42} = x^5 + x^4 + x^3 + x$	$g^{63} = 1$

8.3. Bemenet feldolgozása

8.3.1. Forráskódolás

A forráskódolás az első olyan függvény, amely bemenetként egy karaktert kap és az imént említett véges test egy eleméhez rendeli hozzá, injektíven. Tehát feleltessük meg a 64 karakter mindegyikét a test 64 elemével úgy, hogy egyértelmű legyen a leképezés. Itt már a polinomok helyett a Pythonban tömbökkel reprezentáltam a testelemeket, így minden karakter megfelel egy páronként különböző, hat hosszú bináris számsorozatnak.

$szóköz = [0,0,0,0,0,0]$	$p = [0,1,0,0,1,1]$	$F = [0,0,1,0,0,1]$	$V = [0,0,1,1,0,1]$
$a = [0,0,0,0,1,0]$	$q = [1,0,0,1,1,0]$	$G = [0,1,0,0,1,0]$	$W = [0,1,1,0,1,0]$
$b = [0,0,0,1,0,0]$	$r = [0,0,1,1,1,1]$	$H = [1,0,0,1,0,0]$	$X = [1,1,0,1,0,0]$
$c = [0,0,1,0,0,0]$	$s = [0,1,1,1,1,0]$	$I = [0,0,1,0,1,1]$	$Y = [1,0,1,0,1,1]$
$d = [0,1,0,0,0,0]$	$t = [1,1,1,1,0,0]$	$J = [0,1,0,1,1,0]$	$Z = [0,1,0,1,0,1]$
$e = [1,0,0,0,0,0]$	$u = [1,1,1,0,1,1]$	$K = [1,0,1,1,0,0]$	$= = [1,0,1,0,1,0]$
$f = [0,0,0,0,1,1]$	$v = [1,1,0,1,0,1]$	$L = [0,1,1,0,1,1]$	$! = [0,1,0,1,1,1]$
$g = [0,0,0,1,1,0]$	$w = [1,0,1,0,0,1]$	$M = [1,1,0,1,1,0]$	$? = [1,0,1,1,1,0]$
$h = [0,0,1,1,0,0]$	$x = [0,1,0,0,0,1]$	$N = [1,0,1,1,1,1]$	$. = [0,1,1,1,1,1]$
$i = [0,1,1,0,0,0]$	$y = [1,0,0,0,1,0]$	$O = [0,1,1,1,0,1]$	$, = [1,1,1,1,1,0]$
$j = [1,1,0,0,0,0]$	$z = [0,0,0,1,1,1]$	$P = [1,1,1,0,1,0]$	$; = [1,1,1,1,1,1]$
$k = [1,0,0,0,1,1]$	$A = [0,0,1,1,1,0]$	$Q = [1,1,0,1,1,1]$	$+ = [1,1,1,1,0,1]$
$l = [0,0,0,1,0,1]$	$B = [0,1,1,1,0,0]$	$R = [1,0,1,1,0,1]$	$- = [1,1,1,0,0,1]$
$m = [0,0,1,0,1,0]$	$C = [1,1,1,0,0,0]$	$S = [0,1,1,0,0,1]$	$* = [1,1,0,0,0,1]$
$n = [0,1,0,1,0,0]$	$D = [1,1,0,0,1,1]$	$T = [1,1,0,0,1,0]$	$/ = [1,0,0,0,0,1]$
$o = [1,0,1,0,0,0]$	$E = [1,0,0,1,0,1]$	$U = [1,0,0,1,1,1]$	$\% = [0,0,0,0,0,1]$

A példamondatunkat vigyük tovább és feleltessük meg a karaktereit a megfelelő számsorozatoknak, az alábbi módon:

$T \mapsto [1,1,0,0,1,0]$	$c \mapsto [0,0,1,0,0,0]$	$i \mapsto [0,1,1,0,0,0]$	
$h \mapsto [0,0,1,1,0,0]$	$a \mapsto [0,0,0,0,1,0]$	$s \mapsto [0,1,1,1,1,0]$	
$e \mapsto [1,0,0,0,0,0]$	$t \mapsto [1,1,1,1,0,0]$	$szóköz \mapsto [0,0,0,0,0,0]$	
$szóköz \mapsto [0,0,0,0,0,0]$	$szóköz \mapsto [0,0,0,0,0,0]$	$s \mapsto [0,1,1,1,1,0]$	*stb.*

8.3.2. Blokk-kódolás

Ezek után a blokk-kódolás következik, azonban mivel az algoritmusomat úgy építettem föl, hogy a blokkokra bontást már korábban elvégezte, ezért már csak a kódolás hiányzik. Szóval az üzenet, amit kódolni szeretnénk négy hosszú, ezért a $k = 4$. Ezt követően az N-t kellett meghatároznom, ami a kódszó és az $\underline{\alpha}$ hosszát is definiálja, valamint a hibajavító képessége a kódnak is ettől a számtól függ. A választásom az $N = 8$ lett és így behelyettesítve az egyenlőtlenségbe kijön, hogy 2-hibajavító lesz a kódom. Azonban N-nek bármely k-nál nagyobb számot, ami legfeljebb 64 választhattam volna, ez szabadon meghatározható, de figyelni kell, hogy a későbbiekben jelentősen több számolást kell elvégezni, minél nagyobb az N. Így megvan, hogy mekkora legyen az $\underline{\alpha}$ vektor, de kérdés még, hogy mik legyenek az elemei. A projektemben igyekeztem az $\underline{\alpha}$-t a legegyenletesebben elosztani a testelemeken, tehát az elemei hatványkitevős alakban a következők lettek: $\underline{\alpha} = (g^4, g^{12}, g^{20}, g^{28}, g^{36}, g^{44}, g^{52}, g^{60})$. Ezeken a pontokon értékeljük ki az $f_m(x)$ függvényt, amelyből a $\underline{c}$ vektort, mint kódszót kapjuk.
Lássuk ezt a példánkon keresztül, vegyük csak az első négyes blokkot:

$$\begin{aligned}
f_m([0,1,0,0,0,0]) &= [1,1,0,0,1,0] + [0,0,1,1,0,0]\cdot[0,1,0,0,0,0] + [1,0,0,0,0,0]\cdot[0,1,0,0,0,0]^2 + \\
&\quad + [0,0,0,0,0,0]\cdot[0,1,0,0,0,0]^3 = [1,1,1,1,0,1] \quad \text{(az 1. eleme a kódszónak)} \\
f_m([0,0,0,1,0,1]) &= [0,1,1,0,0,1] + [0,0,0,1,1,0]\cdot[0,0,0,1,0,1] + [0,1,0,0,0,0]\cdot[0,0,0,1,0,1]^2 + \\
&\quad + [0,0,0,0,0,0]\cdot[0,0,0,1,0,1]^3 = [1,1,0,1,1,0] \quad \text{(az 2. eleme a kódszónak)} \\
&\vdots \\
f_m([1,1,1,0,0,1]) &= [0,1,1,0,0,1] + [0,0,0,1,1,0]\cdot[1,1,1,0,0,1] + [0,1,0,0,0,0]\cdot[1,1,1,0,0,1]^2 + \\
&\quad + [0,0,0,0,0,0]\cdot[1,1,1,0,0,1]^3 = [1,1,0,0,1,1] \quad \text{(az utolsó eleme a kódszónak)}
\end{aligned}$$

Az általános lépés a következő: $f_m(\alpha_i) = m_0 + m_1\alpha_i + m_2\alpha_i^2 + m_3\alpha_i^3 = c_i$.
Tehát ha a „T” karaktert szeretnénk kódolni, akkor a test táblázatban a $[1,1,0,0,1,0]$-nak, a testelemek közül pedig a $g^{46} = x^5 + x^4 + x$ -nek felel meg.

A kódszó amit kapunk a következő lesz:
$[[1,1,1,1,0,1],[1,1,0,1,1,0],[1,1,0,1,1,1],[0,1,0,1,0,1],$
$[0,0,1,0,1,1],[0,1,0,1,0,0],[1,1,1,0,0,1],[1,1,0,0,1,1]]$.

Python bemeneti értékek

A program bemenetnek megkapja az elküldeni kívánt üzenetet, mint *message* változót, az ábécét, ami karakterekhez és véges test kitevőihez rendel bináris számsorozatokat (*abc, idx_poly*), valamint az $\underline{\alpha}$ vektort. A meghívott függvények és a teljes kód a függelékben tekinthető meg.

Python implementációja a kódolásnak

```
c_list = []

block_list = block_separator(message)

for block in block_list:
    c = []
    m = source_coding(abc, block)
    for a in alpha:
        c.append(f_m(m, a))
    c_list.append(c)
```

Ebben a kódrészletben először négyes blokkokra osztom az üzenetet a *block_separator* függvény segítségével, majd egy *for* ciklussal egyesével forráskódolom (*source_coding*) majd az *f_m* függvénnyel kódszóvá alakítom őket.

8.4. Zajos csatorna szimulálása

A korábban leírtak alapján, mivel ismert, hogy a csatorna hibáinak jellege nem determinisztikus, ezért szerettem volna random zajjal dolgozni. Ezt a Pythonban lévő random szám generátorral tudtam kivitelezni. A hibák számát egy t számlálóval tartom számon és ennek átírásával tudom növelni illetve csökkenteni a sérült elemek mennyiségét. Ebben a kódban egy hibának egy teljes hat hosszú számsorozat megváltoztatását értjük.
A példánkban szereplő kódszó kapjon mondjuk két hibát. Ekkor lehet látni, hogy a random hiba az első és a hetedik helyen változtatta meg a kódszót az alábbira:
$[[1,1,1,1,1,0],[1,1,0,1,1,0],[1,1,0,1,1,1],[0,1,0,1,0,1],$
$[0,0,1,0,1,1],[0,1,0,1,0,0],[1,1,0,0,1,0],[1,1,0,0,1,1]]$.

Python implementációja a csatornának

```
c_damaged_list = []

for c in c_list:
    t = random.randint(0,2)
    error_list = random.sample(range(1, len(alpha)+1), t)
    c_damaged = damage(c, error_list)
    c_damaged_list.append(c_damaged)
```

Ebben a kódrészletben a c kódszavakból a *damage* fügvénnyel *c_damage* hibás kódszavak lesznek szimulálva a zajos csatornát. Az hogy hány hiba történjen a t random sorsolt változó értéke határozza meg és hogy ezek hol legyenek, azt is visszatevés nélkül sorsoljuk.

8.5. Dekódolás

8.5.1. Hibahely detektálás

Az elméleti összefoglalóban leírt $N - k \geq 2t$ egyenlőtleség teljesülésének ellenőrzését a kódomban nem végzem el, hiszen a valóságot próbálja szimulálni, ahol pedig előre nem prediktálható a sérülések száma a csatornán.
Ezek után a $Q(x) = R(x) \cdot E(x)$ egyenletrendszer ismeret értékeiből konstruálok egy mátrixot, ahol az $E(x) = (x-e_1)(x-e_2) = x^2+(-e_1-e_2)x+e_1e_2$ átalakítással az ismeretlen együttható az $-e_1-e_2 = l_1$ és az $e_1e_2 = l_2$ lesznek. Így a mátrix, amelyre Gauss-eliminációt alkalmazunk, az alábbi lett:

$$\begin{array}{c|cccccccc|c} & q_0 & q_1 & q_2 & q_3 & q_4 & q_5 & l_1 & l_2 & \\ \alpha_1 & 1 & \alpha_1 & \alpha_1^2 & \alpha_1^3 & \alpha_1^4 & \alpha_1^5 & -c_1 & -c_1 \cdot \alpha_1 & c_1 \cdot \alpha_1^2 \\ \alpha_2 & 1 & \alpha_2 & \alpha_2^2 & \alpha_2^3 & \alpha_2^4 & \alpha_2^5 & -c_2 & -c_2 \cdot \alpha_1 & c_2 \cdot \alpha_2^2 \\ \vdots & & & & & \vdots & & & & \vdots \\ \alpha_8 & 1 & \alpha_8 & \alpha_8^2 & \alpha_8^3 & \alpha_8^4 & \alpha_8^5 & -c_8 & -c_8 \cdot \alpha_8 & c_8 \cdot \alpha_8^2 \end{array}$$

A Gauss-elimináció eredményéből sok értékes inormációt kapunk. Először is ha a kódunk a számolás alatt nem futott hibára, akkor valóban két helyen sérült. Továbbá az l_1 valamint az l_2 értékeit felhasználva, az $E(x) = x^2 + l_1x + l_2$ függvényen végigiterálunk az $\underline{\alpha}$ vektor értékeivel és megnézzük, hogy mely α-kra ad nulla értéket. Ebből pontosan kettő lesz és meg is tudtuk, hogy ezen α helyeken sérült meg a kódszó.
Előfordul, hogy kettőnél kevesebb hibát tartalmaz a kódszó, ilyenkor hibára futunk a Gauss-eliminációban, de elkapjuk azt és kezeljük. Mivel az elején feltételeztük, hogy létezik második hiba, így az l_1 illetve l_2 értékekre nem kapunk pontos megoldásokat, hiszen végtelen szabadsági

fokkal rendelkeznek az e_2 tényező miatt. Ezért módosítsuk a mátrixot, amire meghívjuk a Gauss-eliminációt az alábbira:

$$\begin{array}{c|ccccccc|c} & q_0 & q_1 & q_2 & q_3 & q_4 & q_5 & e_1 & \\ \alpha_1 & 1 & \alpha_1 & \alpha_1^2 & \alpha_1^3 & \alpha_1^4 & \alpha_1^5 & -c_1 & -c_1 \cdot \alpha_1 \\ \alpha_2 & 1 & \alpha_2 & \alpha_2^2 & \alpha_2^3 & \alpha_2^4 & \alpha_2^5 & -c_2 & -c_2 \cdot \alpha_1 \\ \vdots & & & & & \vdots & & & \vdots \\ \alpha_8 & 1 & \alpha_8 & \alpha_8^2 & \alpha_8^3 & \alpha_8^4 & \alpha_8^5 & -c_8 & -c_8 \cdot \alpha_8 \end{array}$$

Ennek a megoldásában már megkapjuk az egyetlen hiba pontos helyét, mint e_1 értéket.
Előfordulhat, hogy nem történt hiba. Ebben az esetben ebben a Gauss-eliminációban is hibára futunk. Ekkor megtartjuk a korábban hibásnak feltételezett kódszót, amiről most már tudjuk, hogy nem sérült.
Az utolsó lehetséges eset, amely előállhat az amikor kettőnél több a sérülések száma és emiatt fut hibára a program. Ebből az esetből próbafuttatásokkal az alábbi konklúziókat vontam le: A hiba nem a Gauss-eliminációban keletkezik, mert az gond nélkül lefut, hanem az azutáni részben amikor a megkapott l_1, l_2 értékeket próbálja értelmezni a program. Itt az $\underline{\alpha}$ értékein iterál végig és ez esetben a program egyiket sem találja megfelelőnek. Ebből adódóan kapunk hibát, ráadásul ez azt jelenti, hogy egy hibahelyet sem tudott meghatározni a programkódunk. Lássunk erre egy példát: Tegyük fel, hogy kettő hibát várunk de három érkezik. Ez esetben a hibahely polinomunk az $E(X) = (x - e_1)(x - e_2) = x^2 + (-e_1 - e_2)x + e_1e_2$ helyett az $E(X) = (x - e_1)(x - e_2)(x - e_3) = x^3 + (-e_1 - e_2 - e_3)x^2 + (e_1e_2 + e_1e_3 + e_2e_3)x + (-e_1e_2e_3)$ lenne, ahol $l_1 = (-e_1 - e_2 - e_3), l_2 = (e_1e_2 + e_1e_3 + e_2e_3), l_3 = (-e_1e_2e_3)$. Itt láthatjuk, hogy ez egy harmad fokú hibaegyenlet. A programkód csak egy másodfokú hibaegyenlet megoldásait keresi, ezért ha találna is l_1, l_2 értékeket mindet így sem tudná meghatározni. Azonban az α-kat úgy választottam, hogy egyik sem egyenlő nullával vagy eggyel, ezért az e_3 elem minden esetben módosít az l_1, l_3 értékeken. Ezért nem ad megoldást és fut hibára a programkód kettőnél több sérülés esetén.

8.5.2. Hiba javítás

Kiderült, hogy hol történt a hiba. Ezek után intuitíven az lenne a következő lépés, hogy találjuk ki ezeken a helyeken mi lehetett az eredeti kód. Ehelyett azonban most a sértetlen kódrészletekből következtetjük ki a teljes eredeti üzenetet.
Konstruáljunk még egy mátrixot, amely az $f_m(x) = m_0 + m_1x + m_2x^2 + m_3x^3$ ismert értékeiből áll, de azúttal csak azon $\underline{\alpha}$ értékeket helyettesítsük be, amelyről biztosan tudjuk, hogy nem sérültek meg. Legyen ez a két $\underline{\alpha}$ érték az α_1, és az α_7. Ismét Gauss-eliminációval számoljuk ki a függvény ismeretlen együtthatóit. Ekkor az alábbi mátrixból induljunk ki:

$$\begin{array}{c|cccc|c} & m_0 & m_1 & m_2 & m_3 & \\ \alpha_2 & 1 & \alpha_2 & \alpha_2^2 & \alpha_2^3 & c_2 \\ \alpha_3 & 1 & \alpha_3 & \alpha_3^2 & \alpha_3^3 & c_3 \\ \vdots & & \vdots & & & \vdots \\ \alpha_8 & 1 & \alpha_8 & \alpha_8^2 & \alpha_8^3 & c_8 \end{array}$$

A számolás eredménye megadja az eredeti üzenet tömbökkel reprezentált alakját, ezt átforítva megkapjuk azt a négy karaktert amely az üzenet volt.
A példánkban ez az alábbi:
$[[1,1,0,0,1,0],[0,0,1,1,0,0],[1,0,0,0,0,0],[0,0,0,0,0,0]] \mapsto [T, h, e, \textit{szóköz}]$

8.5.3. Konkatenálás

Miután megérkezett és dekódoltuk az összes négy hosszú karaktert készen áll, hogy ezeket összeragasztva megkapjuk az eredeti üzenet teljes alakját. Visszakaptuk a

The cat is sleeping. The dog is eating.

üzenetet helyesen tehát hibátlan a dekódolás.

Python implementációja a dekódolásnak

```
decoded_message_list = []

for c in c_damaged_list:
    A, number_of_errors = gauss(alpha, c)
    damaged_alpha = error_locator(A, number_of_errors)
    decoded_message_list.append(decoder(alpha,
                                        c,
                                        number_of_errors,
                                        damaged_alpha,
                                        idx_poly))
```

Ebben a kódban bemenetként egy sérült kódszót kapunk (*c_damage*) elvégezzük a Gauss-eliminációt (*gauss*), ebből megtudjuk a hibák számát, amiből az *error_locator* függvény meghatározza a hibák pontos helyét. Innentől a *decoder* függvény már a dekódolt üzenetet adja vissza, ezeket konkatenálva megkapjuk a teljes szöveget, amit a feladó közölni szeretett volna.

Függelék

- A projekt során használt Python kód:

  `https://github.com/sargakerites/BScThesis`

- A kódban használt függvények:

```
def block_separator(message):
    i = 0
    blocks = []
    lista = []
    for m in message:
        if i < 3:
            lista.append(m)
            i += 1
        else:
            lista.append(m)
            blocks.append(lista)
            lista = []
            i = 0
    if lista != []:
        blocks.append(lista)
    while len(blocks[-1]) < 4:
        blocks[-1].insert(3, "␣")
    return blocks

def source_coding(abc, word):
  m = []
  for i in range(len(word)):
    m.append(abc[word[i]])
  return m
```

```
def f_m(m, x):
    c = 0
    x_power = copy.deepcopy(x)
    for i in range(len(m)):
      if i == 0:
        c = m[0]
      else:
        c = F.add(c, F.mul(m[i], x_power))
        x_power = F.mul(x_power, x)
    return c

def damage(c, error_list):
  c_damaged = copy.deepcopy(c)
  for i in error_list:
    c_damaged[i-1] = []
    for j in range(6):
      c_damaged[i-1].append(random.randint(0, 1))
  return c_damaged

def gauss(alpha, c_damaged):
    n = 8
    a = []

    for i in range(n):
        l = []
        for j in range(n+1):
            l.append(0)
        a.append(l)

    for i in range(n):
        for j in range(n+1):
            if j == 0:
                a[i][j] = idx_poly[1]
            elif j > 0 and j <= 5:
                co = alpha[i]
                for k in range(j-1):
                    co = F.mul(co, alpha[i])
                a[i][j] = co
```

```
            elif j == 6:
                a[i][j] = c_damaged[i]
            elif j == 7:
                co = F.mul(alpha[i], c_damaged[i])
                a[i][j] = co
            else:
                co = F.mul(c_damaged[i], F.mul(alpha[i], alpha[i]))
                a[i][j] = co

n = 8
number_of_errors = 2

for i in range(n):
    if a[i][i] == [0, 0, 0, 0, 0, 0] and i != n-1:
        k = 1
        while a[i+k][i] == [0, 0, 0, 0, 0, 0]:
            k += 1
            if i+k > 7:
                number_of_errors = 0
                return a, number_of_errors
        temp = copy.deepcopy(a[i])
        a[i] = copy.deepcopy(a[i+k])
        a[i+k] = temp

    if a[i][i] == 1:
        pass
    else:
        divisor = copy.deepcopy(a[i][i])
        for j in range(n+1):
            if divisor == [0, 0, 0, 0, 0, 0]:
                pass
            else:
                inverse = F.inv(divisor)
                a[i][j] = F.mul(a[i][j], inverse)

    for j in range(n):
        if j == i:
            pass
```

```
            else:
                ratio = a[j][i]
                for k in range(i, n+1):
                    s = F.mul(ratio, a[i][k])
                    a[j][k] = F.sub(a[j][k], s)

    if a[-1][-2] == [0, 0, 0, 0, 0, 0]:
        number_of_errors = 1

        n = 8
        a = []

        for i in range(n):
            l = []
            for j in range(n-1):
                l.append(0)
            a.append(l)

        for i in range(n):
            for j in range(n-1):
                if j == 0:
                    a[i][j] = idx_poly[1]
                elif j > 0 and j <= 4:
                    co = alpha[i]
                    for k in range(j-1):
                        co = F.mul(co, alpha[i])
                    a[i][j] = co
                elif j == 5:
                    a[i][j] = c_damaged[i]
                else:
                    co = F.mul(alpha[i], c_damaged[i])
                    a[i][j] = co

        n = 6

        for i in range(n):
            if a[i][i] == [0, 0, 0, 0, 0, 0] and i != n-1:
                k = 1
```

```
                while a[i+k][i] == [0, 0, 0, 0, 0, 0]:
                    k += 1
                temp = copy.deepcopy(a[i])
                a[i] = copy.deepcopy(a[i+k])
                a[i+k] = temp

            if a[i][i] == 1:
                pass
            else:
                divisor = copy.deepcopy(a[i][i])
                for j in range(n+1):
                    if divisor == [0, 0, 0, 0, 0, 0]:
                        pass
                    else:
                        inverse = F.inv(divisor)
                        a[i][j] = F.mul(a[i][j], inverse)

            for j in range(n+2):
                if j == i:
                    pass
                else:
                    ratio = a[j][i]
                    for k in range(i, n+1):
                        s = F.mul(ratio, a[i][k])
                        a[j][k] = F.sub(a[j][k], s)

    return a, number_of_errors

def error_locator(a, numer_of_errors):
    l = []
    if number_of_errors == 0:
        return l
    elif number_of_errors == 1:
        if a[-3][-1] == [0, 0, 0, 0, 0, 0]:
            return l
        else:
            l.append(a[-3][-1])
            return l
```

```
    else:
        l1 = a[6][8]
        l2 = a[7][8]
        for i in alpha:
            s = F.sub(l2, i)
            if l1 == F.mul(s, i):
                e2 = i
        e1 = F.sub(l2, e2)
        l.append(e1)
        l.append(e2)
        return l

def decoder(alpha, c_damaged, number_of_errors, damaged_alpha, idx_poly):

    alpha_lack = list(copy.deepcopy(alpha))
    c_lack = copy.deepcopy(c_damaged)
    ind = []
    mes = []
    message_back = str()
    abc_key_list = list(abc.keys())
    abc_val_list = list(abc.values())

    for e in damaged_alpha:
        alpha_lack.remove(e)

    for e in damaged_alpha:
        i = alpha.index(e)
        ind.append(i)

    ind.sort(reverse=True)

    for i in ind:
        c_lack.pop(i)

    n = 4
    a = []

    for i in range(n):
```

```
        l = []
        for j in range(n+1):
            l.append(0)
        a.append(l)

    for i in range(n):
        for j in range(n+1):
            if j == 0:
                a[i][j] = idx_poly[1]
            elif j == 1:
                a[i][j] = alpha_lack[i]
            elif j == 2:
                pw = F.mul(alpha_lack[i], alpha_lack[i])
                a[i][j] = pw
            elif j == 3:
                pw = F.mul(alpha_lack[i], alpha_lack[i])
                cu = F.mul(pw, alpha_lack[i])
                a[i][j] = cu
            else:
                a[i][j] = c_lack[i]

    n = 4

    for i in range(n):
        if a[i][i] == [0, 0, 0, 0, 0, 0]:
            k = 1
            while a[i+k][i] == [0, 0, 0, 0, 0, 0]:
                k += 1
            temp = copy.deepcopy(a[i])
            a[i] = copy.deepcopy(a[i+k])
            a[i+k] = temp

        if a[i][i] == [0, 0, 0, 0, 0, 1]:
            pass
        else:
            divisor = copy.deepcopy(a[i][i])
            for j in range(n+1):
                inverse = F.inv(divisor)
```

```
                a[i][j] = F.mul(a[i][j], inverse)

        for j in range(n):
            if j == i:
                pass
            else:
                ratio = a[j][i]
                for k in range(i, n+1):
                    s = F.mul(ratio, a[i][k])
                    a[j][k] = F.sub(a[j][k], s)

    for i in range(len(a)):
        mes.append(a[i][-1])

    mes[0] = F.mul(mes[0], [0, 0, 0, 0, 1, 0])

    for m in mes:
        message_back = message_back + abc_key_list[abc_val_list.index(m)]

    return message_back

def block_concatenator(blocks):
    message = str()
    for block in blocks:
        message = message + block
    return message
```

Irodalomjegyzék

[1] Ashish Choudhury videója:
https://www.youtube.com/watch?v=6X1OCX-iq9w&list=LL&index=4&t=1s

[2] Kiss Emil. Bevezetés az algebrába. Typotex Kft, 2007.

[3] Györfi László, Győri Sándor és Vajda István. Információ-és kódelmélet. Typotex Kft, 2000.

[4] Wettl Ferenc. Kódelmélet és kriptográfia. BME, 2011

[5] Szőnyi Tamás honlapja: https://szonyitamas.web.elte.hu/code.html

[6] Ivanyos Gábor jegyzete: https://math.bme.hu/~ig/kod/vazlat.pdf

[7] https://www.mathreference.org/index/page/id/433/lg/hu

[8] Lovász László, Pelikán József és Vesztergombi Katalin. Diszkrét matematika. Typotex Kft, 2010.

[9] Montágh Balázs, Bérczi Gergely, Gács András, Szőnyi Tamás, Ambrus Gergely, Lovász László, Wettl Ferenc, Hraskó András, Schmidt Edit, Gyárfás András, Tóth Géza, Pach János, Laczkovich Miklós, Frenkel Péter, Kiss Emil, Reiman István, Pelikán József, Csikós Balázs, Moussong Gábor, Szűcs András és Recski András. Új matematikai mozaik. Typotex Kft, 2003.