NYILATKOZAT

Név: Alexy Marcell

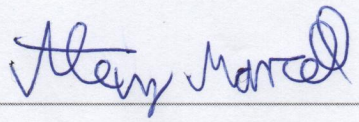
ELTE Természettudományi Kar, szak: Matematika BSc

NEPTUN azonosító: EVOTEW

Szakdolgozat címe:
Reed-Solomon kódok dekódolása

A **szakdolgozat** szerzőjeként fegyelmi felelősségem tudatában kijelentem, hogy a dolgozatom önálló szellemi alkotásom, abban a hivatkozások és idézések standard szabályait következetesen alkalmaztam, mások által írt részeket a megfelelő idézés nélkül nem használtam fel.

Budapest, 2021. 05. 31.



a hallgató aláírása

Reed-Solomon kódok dekódolása

Diplomamunka

Írta: Alexy Marcell

Matematikus szak

Témavezető:

Szőnyi Tamás, egyetemi tanár
Számítógéptudományi Tanszék
Eötvös Loránd Tudományegyetem, Természettudományi Kar

Eötvös Loránd Tudományegyetem
Természettudományi Kar
2021

Köszönetnyilvánítás

Szeretném megköszönni Szőnyi Tamásnak a kódelméletből tartott konzultációkat, a témavezetés elvállalását és a témához kapcsolódó cikkek és egyéb érdekes források bemutatását.

Tartalomjegyzék

1. fejezet

Bevezető

A kommunikáció alapvető emberi igény. Már az ókor óta szerettünk volna információt küldeni egymásnak, jeleket továbbítani a többiek felé, akár nagy távolságokon és hosszú időtartamon keresztül. Ugyanakkor nem minden jel fog pontosan megérkezni a célszemélyhez, útközben keletkezhetnek hibák az elküldött üzenetben, ezért a hibák javítására valamilyen megoldást kell találni.

Jeleket sok csatornán keresztül küldhetünk. Egy üzenet érkezhet hang útján, füstjelként, írott levél formájában, elektromos impulzusként vagy valamilyen egyéb módon. Amennyiben van gyors kétirányú kommunikáció a felek között, a kommunikációs csatornában keletkező hibákat lehet javítani az üzenetek újra küldésével, azonban ehhez szükség van a hiba felismerésére.

Ha a szabályos üzenetek jól elkülönülnek egymástól, akkor nagy valószínűséggel észre lehet venni, hogy történt-e hiba a kommunikációban. Például beszéd közben az emberi hangok sorozatainak csak kis része áll össze mondatokkáa nyelvtani és értelmezési szabályok miatt, ezért zaj fellépése esetén értelmetlenné válhat az üzenet, és vissza lehet kérdezni. Számítógépes rendszerek közti kommunikáció során is le lehet fektetni valamilyen szabályrendszert a küldhető üzenetekre, hogy az összes küldhető üzenet valamilyen kódban legyen. Ilyen szempontból érdekes vizsgálni a kódok hibafelismerő képességét.

Azonban nem mindig adódik lehetőség újrakérni egy üzenet. Előfordulhat, hogy az üzenetet nagy távolságon át küldik, például egy merevlemezt küldenek postán

keresztül, és az újraküldés sok időt venne igénybe. Esetleg lehet, hogy egy információt hosszú ideig tárolnánk, és a tárolás során keletkezik valami hiba benne, például egy CD lemezen tárolt adat sérül, mivel a műanyag burkolat elkezd bomlani. Valamint lehet, hogya küldőnek csak egy lehetősége lesz az adatokat elküldeni, például bizonyos űrkutatási műholdaknak csak egy lehetősége van egy kísérlet elvégzésére ésa mérés eredményének elküldésére. Ilyen esetben érdemes foglalkozni azzal, hogy a küldött üzenetben ne csak a hibákat ismerjük fel, hanem az is, hogy hogyan javítsuk keletkezett hibák egy részét.

A Reed-Solomon kódok jó hibajavítási és hibajelzési paraméterekkel rendelkeznek a többi lehetséges kódhoz képest, ezért a szakdolgozatomban megvizsgáltam egy módszert a Reed-Solomon kódok dekódolására.

1.1. Jelölések

1.1.1. Definíció (Alsó egész rész)**.** Jelölje $\lfloor x \rfloor$ azt a legnagyobb $n \in \mathbb{Z}$ egész számot, melyre $n \leq x$ teljesül. Ezt a függvényt alsó egész résznek nevezem.

Hasonló módon lehet definiálni a **felső egészrészt**, mint a legkisebb olyan egész szám, ami nagyobb x-nél, és amelyet $\lceil x \rceil$ jelekkel szoktak jelölni.

1.1.2. Definíció (Első n szám halmaza)**.** A leírás egyszerűsítése miatt jelölje $[n] = \{1, 2, ..., n\}$ az első n pozitív egész szám halmazát. Ekkor például az $i \in [n]$ jelölés alatt azt értem, hogy az i egy egész szám $1 \leq i \leq n$ között.

A kódolások során általában az elküldendő üzenetet **u**-val, az elküldött üzenetet **w**-vel, a beérkezett (akár hibás) üzenetet **y**-nal fogom jelölni.

2. fejezet

Kódelméleti alapfogalmak

A kódelméleti cikkekben sokszor ugyanazon fogalmak jelölésére eltérő betűket használnak. A dolgozatomban a jelöléseket Ivanyos Gábor Kódelmélet jegyzete [8] és Szőnyi Tamás Szimmetrikus struktúrák [11] online jegyzet alapján fogom bevezetni.

2.1. Kód definíciója

Ahhoz, hogy részletesebben tudjunk beszélni a kódokról, először meg kell adnunk a matematikai definíciójukat. A kódok leírásához meg kell adni egy kódábécét.

2.1.1. Definíció. Legyen adva egy F ábécé véges sok karakterrel. Ezt fogjuk nevezni a **kódábécének**.

2.1.2. Definíció. Jelöljük az **kódábécé elemszámát** $q = |F|$ betűvel.

2.1.3. Definíció. Legyen adva egy n természetes szám. Ezt fogjuk nevezni a **kódhossznak**.

Ezután rátérhetünk a kódokat definiálására.

2.1.4. Definíció. Kódnak nevezzük, és C-vel szoktuk jelölni egy $C \subset F^n$ részhalmazát, ahol F^n az összes n hosszú F elemeiből álló karakterlánc.

A kódoknak további vizsgálatához érdemes bevezetni néhány definíciót:

2.1.5. Definíció. A kódban szereplő elemeket $\mathbf{c} \in C$ **kódszavaknak** nevezzük.

2.1.6. Definíció. A kódban szereplő kódszavak számát nevezzük a **kód méretének**, amelyet $M = |C|$ betűvel jelölünk.

2.1.7. Definíció. A **kód dimenziójának** a $k = \log_q(M)$ számot jelöljük.

2.1.8. Definíció. Az **kódsebesség** vagy **információs ráta** alatt a $R = k/n$ arányt értjük.

2.1.9. Definíció. Hamming-távolság: $d(\mathbf{a}, \mathbf{b}) = |\{i : i \in [n], a_i \neq b_i\}|$. Két kódszó Hamming-távolságán azt a számot értjük, amely megadja, hogy hány darab pozícióban van különböző karakter két kódszóban.

2.1.10. Tétel. *A Hamming-távolság metrika. Vagyis:*

- $d(\mathbf{a}, \mathbf{b}) \geq 0$ *minden* $\mathbf{a}, \mathbf{b} \in C$ *kódszavak esetében.*
- $d(\mathbf{a}, \mathbf{b}) = 0$ *akkor és csak akkor, ha* $\mathbf{a} = \mathbf{b}$.
- $d(\mathbf{a}, \mathbf{b}) = d(\mathbf{b}, \mathbf{a})$ *minden esetben.*
- $d(\mathbf{a}, \mathbf{b}) + d(\mathbf{b}, \mathbf{c}) \geq d(\mathbf{a}, \mathbf{c})$ *minden* $\mathbf{a}, \mathbf{b}, \mathbf{c} \in C$ *kódszavak esetén.*

Bizonyítás:

- Mivel a Hamming-távolságot egy legfeljebb n elemű halmaz elemszámával definiáltuk, ezért két kódszó Hamming-távolsága mindig egy nemnegatív természetes szám lesz.
- Ha két kódszó nem azonos, akkor valamelyik pozícióban eltérő karakterek vannak, és ezért a kódszavak távolsága legalább 1. Tehát csak akkor 0 a távolság, ha azonos a két kódszó.
- A definícióból következik a szimmetria.

- Menjünk végig sorba a kódszavak karakterein, mind az n pozícióban. Ha $a_i \neq c_i$, akkor az $a_i \neq b_i$ és $b_i \neq c_i$ feltételek közül legalább az egyiknek teljesülnie kell. Ezért összesen legalább annyi darab különbözőségi hely van összesen $d(\mathbf{a}, \mathbf{b})$-ban és $d(\mathbf{b}, \mathbf{c})$-ben, mint amennyi különbözőség van $d(\mathbf{a}, \mathbf{c})$-ben.

2.1.11. Definíció. Egy C kód **minimális távolságán** (d_C), azt a legkisebb Hamming-távolságot értjük, amelyet két különböző kódszó között létrejöhet, vagyis $d_C = \min_{a,b \in C, a \neq b} d(a, b)$. Ha egyértelmű, hogy melyik kódról van szó, akkor az alsó indexet el lehet hagyni.

2.1.12. Tétel (Singleton-korlát)**.** *Bármely C kód esetén a méretére teljesül, hogy $M \leq q^{n-d+1}$ vagy $k \leq n-d+1$, ahol M a kód mérete, k a kód dimenziója, d pedig a kód minimális távolságát jelöli.*

Bizonyítás: Amennyiben $d = 1$, és n egy tetszőleges természetes szám, látható, hogy q^n darabnál több kódszó nem lehet, mivel összesen ennyi F^n-beli elem van.

Tegyük fel, hogy egy C kódban a minimális távolság $d \geq 2$. Ezen kód alapján hozzuk létre a C' kódot úgy, hogy kitöröljük az utolsó $d-1$ karaktert az összes kódszóból. Két tetszőleges kódszó távolsága ekkor legalább egy lesz, ezért $\mathbf{c_1} \neq \mathbf{c_2} \in C$ esetén a keletkező $\mathbf{c_1'}, \mathbf{c_2'} \in C'$ kódszavak különbözni fognak. A keletkezett C' kód hossza $n-d+1$, a minimális távolsága legalább 1, és a kód dimenziója változatlan, mivel ugyanannyi kódszót tartalmaz C és C'. Viszont ekkor C' dimenziója nem lehet nagyobb, mint a kód hossza, ezért $k \leq n-d+1$, és ezt átrendezve $k \leq n-d+1$ egyenlőtlenséget kapjuk.

2.1.13. Definíció. MDS kód vagy **maximális távolságú** kódnak nevezzük az olyan kódokat, melyekre a Singleton korlát egyenlőséggel teljesül.

2.2. Lineáris kódok

Sok esetben lehet az F kódábécé karaktereit úgy is lehet tekinteni, mintha egy $\mathbb{F}$ véges test elemei lennének. Ekkor a kódot lehet úgy értelmezni, mint az $\mathbb{F}^n$

vektortér része lenne, és a kódszavakat lehet $\mathbb{F}^n$-beli vektorokként is értelmezni. Kódelméletben ezeket a vektorokat általában sorvektorokként szokták írni.

2.2.1. Definíció. Egy C kódot $[n, k, d]$ **lineáris kódnak** nevezünk, ha $C \subset \mathbb{F}^n$, valamint nemcsak részhalmaza a vektortérnek, hanem egy k dimenziós altere is. d az altér elemei közti minimális Hamming-távolságot jelöli.

2.2.2. Állítás. *Ha C egy lineáris kód, akkor a kód dimenziója megegyezik az altér dimenziójával.*

Bizonyítás: Amennyiben C egy lineáris kód, akkor a kód mérete megegyezik a hozzá tartozó altérnek az elemszámával. Egy k' dimenziós altér elemszáma az $\mathbb{F}_q$ test fölött $q^{k'}$. Emiatt a kód dimenziójára, $(k = \log_q |C|)$, igaz, hogy $k = \log_q q^{k'} = k'$, tehát az altér dimenzió és a kód dimenzió megegyezik.

2.2.3. Definíció. Legyen $C \subset \mathbb{F}^n$ egy n kódhosszal rendelkező lineáris kód . A $\mathbf{c} \in C$ kódszó **Hamming-súlyának** nevezzük az 0-vektortól való Hamming-távolságát. $(w(\mathbf{c}) = d(\mathbf{c}, \mathbf{0}))$

2.2.4. Definíció. Egy C lineáris kód **generátor mátrixa** az a G mátrix, amelynek sorai a kód alterének egy bázisát adják ki.

Ezt a mátrixot fel lehet használni egy $\mathbf{u} \in \mathbb{F}^k$-beli üzenet C-be történő kódolásra, mégpedig a $\mathbf{u}$-hez tartozó $\mathbf{c}$ kódszó: $\mathbf{c} = \mathbf{u}G$. Mivel G sorai bázist alkotnak, ezért az $G : \mathbb{F}^k \to C \subset \mathbb{F}^n$ leképezés injektív, és ezért különböző $\mathbf{u_1}, \mathbf{u_2}$ üzenetekhez különböző $\mathbf{c_1}, \mathbf{c_2}$ kódszavak fognak tartozni.

Egy adott C kódhoz több G generátor mátrix is tartozik. Ugyanakkor, mivel ezek a mátrixok mind ugyanabból a vektorból képeznek injektíven ugyanabba a képtérbe, nem különböznek sokban egymástól. Amennyiben G és G' ugyanahhoz a C kódhoz tartozó generátormátrixok, létezik egy olyan $B \in \mathbb{F}^{k \times k}$ reguláris mátrix, amellyel $G = BG'$.

Ha egy kód koordinátáit felcseréljük, akkor a definíció szerint más kódot kapnánk, mivel a kódszavak halmaza megváltozhat. Hasonlóan, ha egy testelemmel szorzunk egy koordinátát, akkor egy másik kódot kapnánk, de a kód szerkezete lényegileg nem változik meg. Ezért bevezethetünk egy ekvivalenciát a kódok között.

2.2.5. Definíció. Két kódot **ekvivalensnek** tekintünk, ha az egyik kódból a másik megkapható egy monomiális mátrixszal való szorzás eredményeként. A monomiális mátrix egy olyan mátrix, amely minden sorában és oszlopában pontosan egy nem nulla elem van.

2.2.6. Definíció. Egy C lineáris kód **(paritás-)ellenőrző mátrixának** azt a $H \in \mathbb{F}^{(n-k)\times n}$ mátrixot nevezzük, amely transzponáltjának a magja megegyezik a C alterével. Vagyis: $C = \ker H^T$

A ellenőrző mátrixot egy homogén lineáris egyenlőségrendszerként is értelmezhetjük, mivel minden $\mathbf{c} \in \mathbb{F}^n$ vektor esetén $\mathbf{c}H^T = \mathbf{0}$ pontosan akkor teljesül, amikor $\mathbf{c} \in C$ kódszó.

Lineáris alterek vizsgálata során érdekes kérdés az, hogy az altérre merőleges vektorok hol helyezkedik el, milyen szerkezetűek. Lineáris kódok esetén érdemes vizsgálni, hogy milyen kódot alkot a merőleges vektorokból álló halmaz.

2.2.7. Definíció. Egy C **lineáris kód duálisának** hívjuk és $C^\perp$-vel jelöljük azon $\mathbb{F}^n$-beli vektorok halmazát, amelyek merőlegesek a C kód által generált altérre.

A merőleges alterekre vonatkozó lineáris algebrai állítások alapján a következő állítást kapjuk:

2.2.8. Állítás. *A $C^\perp$ duális kód dimenziója $n - k$. Az eredeti kód ellenőrző és generátor mátrixa a duális kód generátor és ellenőrző mátrixa lesz. ($G_{C^\perp} = H_C$ és $H_{C^\perp} = G_C$)*

Lineáris kódok MDS tulajdonságainak a vizsgálatához a szisztematikusság definícióját vezessük be.

2.2.9. Definíció. Egy C lineáris kód $[n, k]$ paraméterekkel **szisztematikus** az $i_1, i_2, ..., i_k$ helyen, ha a kódszavakat ezen koordinátákra levetítve megkaphatjuk az összes $\mathbb{F}^k$-beli szót.

2.2.10. Állítás. *Egy C lineáris kód pontosan akkor szisztematikus az $i_1, i_2, ..., i_k$ helyen, ha bármely generátor mátrixának ezen indexhez tartozó oszlopai lineárisan függetlenek.*

Bizonyítás: Feltehetjük, hogy az $i_1, i_2, ..., i_k$ indexek az első k indexet jelölik. Legyen $G_0 \in \mathbb{F}^{k\times k}$ egy G generátormátrix első k oszlopából álló részmátrixa. A G_0 mátrixnak a képtere pontosan akkor lesz az egész $\mathbb{F}^k$ tér, vagyis C pontosan akkor lesz szisztematikus az $i_1, i_2, ..., i_k$ koordinátákon, ha G_0 reguláris mátrix, vagyis ha az oszlopai lineárisan függetlenek.

2.2.11. Állítás. *Ha egy C kód szisztematikus az $i_1, i_2, ..., i_k$ helyen, akkor a $C^\perp$ kód szisztematikus lesz a maradék helyen.*

Bizonyítás: Feltehetjük, hogy a C kód az első k indexen szisztematikus. Vegyük egy G generátor és egy H ellenőrző mátrixát. Jelölje G_0 az első k, G_1 az utolsó $n-k$ oszlopából álló részmátrixait G-nek. Hasonlóan bontsuk fel a H mátrixot is a H_0 és H_1 részmátrixokra. Tegyük fel, hogy a H_1 mátrix oszlopai lineárisan összefüggnek. Ekkor van egy olyan $\mathbf{c} \in C$ kódszó, amelynek az első k koordinátája 0, de az utolsó $n-k$ koordináta nem mindegyike nulla. Másfelől az összes $\mathbf{c}$ kódszót megkaphatjuk $\mathbf{c} = \mathbf{u}G$ alakban valamilyen $\mathbf{u} \in \mathbb{F}^k$ vektorral. Mivel G_0 reguláris, ezért ha $\mathbf{c}$ első k koordinátája 0, akkor $\mathbf{u} = \mathbf{0}$. Ugyanakkor ha $\mathbf{c} = \mathbf{u}G$ akkor $\mathbf{u}G = \mathbf{0}G = \mathbf{0}$, és ez ellentmondásban van azzal, hogy az utolsó $n-k$ koordináta nem mindegyike 0, tehát igaz az állítás.

2.2.12. Állítás. *Egy C lineáris kód $[n, k]$ paraméterekkel pontosan akkor MDS tulajdonságú, ha bármely k indexében szisztematikus lesz.*

Bizonyítás: Ha egy C kód bármely k indexében szisztematikus, akkor bármely $n-k$ koordináta kitörlése után megkaphatnánk az összes $\mathbb{F}^k$-beli vektort, ezért a kód minimális távolsága legalább $n-k+1$, és emiatt MDS.

Másfelől ha egy C lineáris kód MDS tulajdonságú, akkor $d = n-k+1$, emiatt valamely $n-k$ koordináta kitörlése után C' kód egy k dimenziós és k hosszú kód lesz, ami csak a teljes $\mathbb{F}^k$ tér lehet, és ezért a kód szisztematikus lesz bármely k koordinátájában.

2.2.13. Állítás. *Ha egy C kód MDS tulajdonságú, akkor a duális kód is MDS lesz.*

Bizonyítás: Ha a C kód MDS tulajdonságú, akkor bármely k-helyen szisztematikus lesz. Emiatt a duális $C^\perp$ kód bármely $n-k$ helyen szisztematikus lesz, ami miatt $C^\perp$ MDS.

2.3. Hibajavítás és jelzés

2.3.1. Definíció. Egy $\mathbf{y}$ üzenet t **hibát tartalmaz**, ha egy eredetileg elküldött $\mathbf{c}$ kódszóhoz képest $\mathbf{y}$ Hamming-távolsága t.

2.3.2. Definíció. Egy algoritmus t**-hibajavító** a C kódra nézve, ha egy $\mathbf{c} \in C$ kódszótól egy legfeljebb t Hamming-távolságra lévő $\mathbf{y} \in \mathbb{F}^n$ üzenet alapján vissza tudja fejteni az eredeti $\mathbf{c}$ kódszót.

2.3.3. Definíció. Egy algoritmus t**-törlésjavító** tulajdonságú a C kódra nézve, ha egy $\mathbf{c} \in C$ kódszóból t' darab előre ismert helyen való törléssel keletkezett $\mathbf{y} \in \mathbb{F}^{n-t'}$ üzenet alapján vissza tudja fejteni az eredeti $\mathbf{c}$ kódszót.

2.3.4. Definíció. Egy algoritmus t**-hibafelismerő** tulajdonságú a C kódra nézve, ha egy $\mathbf{c} \in C$ kódszóból t darab hiba keletkezése után a keletkezett $\mathbf{y} \in \mathbb{F}^n$ üzenetről meg tudja állapítani, hogy nem az eredetileg küldött üzenet.

Egy egyszerű hibajavító algoritmus az, hogy egy adott $\mathbf{y}$ üzenet esetén minden $\mathbf{c} \in C$ kódszóra megnézzük $\mathbf{y}$ Hamming-távolságát $\mathbf{c}$-től, és kiválasztjuk a legközelebbit ezek közül. Ez az algoritmus legfeljebb $t = \lfloor (d-1)/2 \rfloor$ hibát tud javítani, ugyanis több hiba esetén már nem feltétlenül az eredeti $\mathbf{c}$ kód lesz az egyedüli legközelebbi kód $\mathbf{y}$-hez.

Hasonló módon lehet törlésjavító algoritmust is készíteni, amely legfeljebb $t = d - 1$ törlést képes javítani, ugyanis több törlés esetén már lehetséges, hogy egy adott $\mathbf{y}'$ üzenet több különböző $\mathbf{c_1}, \mathbf{c_2}$ kódszóból is meg lehet kapni.

Hiba felismeréséhez egyszerűen felsorolhatjuk a kód összes szavát, és megnézhetjük, hogy a kapott üzenet benne van-e a felsoroltak közt.

Ha egy algoritmus egy üzenetben több, mint $t = \lfloor (d-1)/2 \rfloor$ hibát is tud javítani, akkor nem feltétlenül tud mindig pontosan egy kódszót találni, lehetséges, hogy kódszavak egy listáját adja vissza eredményként.

Van a t hiba-, törlésjavító, és t hibafelismerő tulajdonságot nemcsak algoritmusokra, hanem kódokra is szokták alkalmazni. Ekkor ezzel azt jelölik, hogy hány darab hiba vagy törlés esetén van egyértelmű visszafejtés. A fennt leírt gondolatmenet alapján a következő állítást kapjuk:

2.3.5. Állítás. *Egy d minimális távolsággal rendelkező C kód $t = \lfloor (d-1)/2 \rfloor$* ***hibajavító*** *illetve $t = d - 1$* ***törlésjavító és hibafelismerő*** *tulajdonságú.*

Egy kód javító képessége tehát szoros összefüggésben van a minimális kódtávolsággal. Ha ehhez hozzávesszük a Singleton-korlátot, akkor a következő állítást kapjuk:

2.3.6. Állítás. *Egy MDS tulajdonságú C kódnál nincs nagyobb javító tulajdonsággal rendelkező kód az azonos kódhosszal és kódmérettel rendelkező kódok közt.*

Bizonyítás: Legyenek adva az C_1 MDS kód n kódhosszal és M kódmérettel, és t hibajavító képességgel, vagyis $t = \lfloor (d_{C_1} - 1)/2 \rfloor$, ami akkor teljesül, ha $2t + 1 \leq d_{C_1} \leq 2t + 2$. Tegyük fel, hogy a C_2 kód azonos hossz és méret mellett $t + 1$ hibajavításra képes. Ekkor a Singleton korlát és a fenti állítás miatt: $2(t+1)+1 \leq d_{C_2} \leq n-k+1$. Ugyanakkor az MDS tulajdonság miatt $n-k+1 = d_{C_1}$, és ekkor $d_{C_1} \leq 2t + 2 < 2t + 3 \leq n - k + 1 = d_{C_1}$, ami ellentmondás.

2.4. Példák kódokra

A definíciók jobb ismertetésére a következőkben bemutatok néhány egyszerűbb kódot a paramétereikkel együtt.

2.4.1. Ismétlő kód

Az ismétlő kód az egyik legegyszerűbb kód, amely rendelkezik hibajavító képességgel. A m-szeres ismétlőkódban való kódoláshoz az eredeti üzenetben minden karaktert megismétlünk m-szer, és így megkapjuk a kódolt üzentet. Ilyen módon az ismétlőkód $m - 1$ törlést és $\lfloor \frac{m-1}{2} \rfloor$ hibát tud javítani.

Amennyiben a kódábécé egy $\mathbb{F}$ test, az ismétlőkódot lineáris kódként is lehet értelmezni. Ha $m = 5$, akkor például a következő generátor illetve ellenőrző mátrixok jellemzik a kódot:

$$G = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

$$H = \begin{pmatrix} 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & 1 & -1 \end{pmatrix}$$

Egy egyszerű dekódoló algoritmus az, ha a kapott üzenetet felosztjuk m hosszú szakaszokra, és minden szakaszból kiválasztjuk a legtöbbször előforduló karaktert.

2.4.2. Paritáskód

A paritáskód vizsgálatához először a q elemből álló F kódábécé elemeit meg kell feleltetnünk egy kommutatív csoport elemeinek, ami lehet például egy ciklikus csoport, vagy egy véges test additív csoportja; a csoportműveletet nevezzük összeadásnak. A paritáskód kódolásához először vegyünk $n-1$ bemeneti karaktert; ezeket a karaktereket a kódban az első $n-1$ helyre be fogjuk rakni. Az utolsó karakter megkapásához váltsuk át a bemeneti karaktereket adjuk össze. Jelölje u' azt a csoportelemet, amellyel ki kell egészíteni az előző összeget úgy, hogy az egységelemet kapjunk. Az u' csoportelemhez tartozó F-beli karakter fog a kódszóban az n. helyen fog szerepelni.

Ez a kód nem tud hibát javítani, de egy hibát tud jelezni/egy törlést tud javítani.

Ha a kódábécé mérete $q = p^k$ valamilyen p prímszám estén, akkor a kódot lehet értelmezni az $\mathbb{F}_q$ feletti lineáris kódként is, amelyhez $m = 5$ esetén a következő generátor és ellenőrző mátrixok tartoznak:

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

$$H = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

A generátor és ellenőrző mátrixok alapján lehet látni, hogy az m hosszú paritáskód és ismétlőkód a duálisai egymásnak.

2.4.3. Hamming[7,4,3] kód

A Hamming[7,4,3] kódokat az $\mathbb{F}_2$ test felett definiálhatjuk, $n = 7$ karakter hosszú, és $k = 4$ dimenziós a kódhoz tartozó hozzá tartozó altér. $\mathbb{F}_2^7$-ben.

A kódhoz a következő generátor illetve ellenőrző mátrix tartozik:

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

$$H = \begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}$$

A Hamming[7,4,3] kód három hibát tud jelezni és egy hibát tud javítani.

A hiba javítását elvégezhetjük a ellenőrző mátrix segítségével. Legyen a beérkező üzenet az $\mathbf{y} \in \mathbb{F}_2^7$ vektor. Ez a beérkezett üzenet tartalmaz egy hibát. A hiba alapján az $\mathbf{y}$ vektort felbonthatjuk az eredeti üzenet és a hiba összegére, $\mathbf{y} = \mathbf{w}+\mathbf{e}$, ahol $\mathbf{w}$ egy kódszó és $\mathbf{e}$ egy legfeljebb egy Hamming-súlyú $\mathbb{F}_2^7$-beli vektor.

Vizsgáljuk meg az $\mathbf{z} = \mathbf{y}H^T$ szorzatot. A felbontás alapján $\mathbf{z} = (\mathbf{w}+\mathbf{e})H^T = 0+\mathbf{e}H^T$. Mivel az $\mathbf{e}$ Hamming súlya legfeljebb 1, ezért a $\mathbf{z}$ vagy az azonosan 0 vektor (ekkor nem hibás a kapott üzenet), vagy meg fog egyezni H mátrix egyik oszlopával. Vegyük észre, hogy a H mátrix oszlopai mind különbözőek, ezért a nem 0 vektor esetén az $\mathbf{z}$ egyértelműen meghatároz egy oszlopot H-ból, és az ahhoz az oszlophoz tartozó bit lesz hibás.

3. fejezet

Reed-Solomon kódok

Habár a fenti kódok sok esetben alkalmazhatóak, csak bizonyos paraméter tartományokon működnek. A Reed-Solomon kódok ezzel szemben sokkal tágabb keretek közt alkalmazhatóak, megfelelő véges test választása mellett.

A Reed-Solomon kódokat akkor lehet értelmezni, ha a kódábécé egy $\mathbb{F}$ véges test. Ezen test elemei közül válasszunk ki n különböző elemet, amelyet $x_1, x_2, ..., x_n \in \mathbb{F}$ karakterekkel jelölünk. A bemenő k karakter kódolásához vegyünk egy $k-1$ fokú $f(x)$ polinomot, ahol a polinom együtthatói a bemenő kód elemei. Ezután $x_1, x_2, ..., x_n$ helyen kiértékeljük a polinomot, így megkapva az $y_1 = f(x_1)$, $y_2 = f(x_2)$, ..., $y_n = f(x_n)$ testelemeket. Ez az n darab $y_1, y_2, ..., y_n$ karakter lesz kódolás eredménye. Az általánosított Reed-Solomon kódoknál a veszünk még n darab $v_1, v_2, ..., v_n \in \mathbb{F} \setminus \{0\}$ nemnulla testelemet, és a kiértékelés eredményeit megszorozzuk ezekkel, vagyis $y'_i = v_i f(x_i)$ alakúak lesznek az eredmények. Ez pontosabban definiálva:

3.0.1. Definíció (Általánosított Reed-Solomon (GRS) kód)**.** Legyen adva egy $\mathbf{x} \in \mathbb{F}^n$ vektor $\mathbb{F}$ felett, amelyre teljesül, hogy $x_i = x_j$ pontosan akkor, ha $i = j$, ezen felül legyen még adva egy $\mathbf{v} \in (\mathbb{F} \setminus \{0\})^n$ nemnulla testelemekből álló vektor. Ekkor az ehhez tartozó $k \leq n$ dimenziós Általánosított Reed-Solomon kód:

$$GRS_k(\mathbf{x}, \mathbf{v}) = \{(v_1 f(x_1), v_2 f(x_2), ..., v_n f(x_n)) : f \in \mathbb{F}[x], \deg f < k\} \subset \mathbb{F}^n$$

Megjegyzés: A Reed-Solomon kódok primitív definíciójában, hogy választunk egy $g \in \mathbb{F}$ primitív elemet, (olyan testelem, melynek hatványaiként megkapható

az összes többi nemnulla testelem), a kiértékelési helyeknek az $x_1 = 1$, $x_2 = g$, $x_3 = g^2, ..., x_n = g^{n-1}$ helyeket választjuk, és a kiértékelés a kódolás eredménye, (vagyis $v_i = 1$ minden i-re).

Sokszor nem az összes általánosított Reed-Solomon kódot kell elemezni, hanem elég csak az olyan $GRS_k(\mathbf{x}, \mathbf{v})$ kódokat vizsgálni, amely esetében $\mathbf{v}$ az azonosan egy vektor. Ilyen esetben G betűt el szokták hagyni, és $RS_k(\mathbf{x})$ kódként hivatkoznak erre a kódra.

3.0.2. Definíció (Reed-Solomon (RS) kód)**.**

$$RS_k(\mathbf{x}) = \{(f(x_1), f(x_2), ..., f(x_n)) : f \in \mathbb{F}[x], \deg f < k\} \subset \mathbb{F}^n$$

Habár ez a kód bármilyen $\mathbf{v} \in (\mathbb{F} \setminus \{0\})^n$ vektor választása mellett ekvivalens lesz a $GRS_k(\mathbf{x}, \mathbf{v})$ kóddal, de a duális kódok kezelése miatt érdemes bevezetni az általánosabb definíciót.

3.1. Reed-Solomon kódok MDS tulajdonsága

3.1.1. Tétel. *Az általános Reed-Solomon kód ($GRS_k(\mathbf{x}, \mathbf{v})$) egy lineáris kód az $\mathbb{F}^n$ vektortérben.*

A definíció alapján be lehet látni, hogy az általánosított Reed-Solomon kódok lineáris alterei lesznek az $\mathbb{F}^n$ vektortérnek. Vegyünk $\mathbf{x}$ és $\mathbf{v}$ paraméterekkel egy $GRS_k(\mathbf{x}, \mathbf{v})$ kódot. Ebből a kódból vegyünk két $\mathbf{c_1}.\mathbf{c_2} \in GRS_k(\mathbf{x}, \mathbf{v}) \subset \mathbb{F}^n$ kódszót. Definíció szerint létezik olyan f_1 és f_2 legfeljebb k fokú polinom, hogy $j \in \{1, 2\}$ esetén $c_j = (v_1 f_j(x_1), v_2 f_j(x_2), ..., v_n f_j(x_n)$. Vizsgáljuk meg az $f_3 = f_1 + f_2$ polinomot. Mivel f_1 és f_2 foka is kisebb k-nál, ezért f_3 foka is kisebb lesz. Emiatt létezni fog egy hozzá tartozó $\mathbf{c_3} \in GRS_k(\mathbf{x}, \mathbf{v})$ kódszó. Vizsgáljuk meg minden $i \in [n]$ koordinátára ezt a kódszót. Definíció alapján $(\mathbf{c_3})_i = v_i f_3(x_i) = v_i(f_1 + f_2)(x_i) = = v_i f_1(x_i) + v_i f_2(x_i) = (\mathbf{c_1})_i + (\mathbf{c_2})_i$. Ez minden koordinátára teljesül, és ezért $\mathbf{c_3} = \mathbf{c_1} + \mathbf{c_2}$ igaz lesz, tehát két kódszó összege is benne van a kódban. Hasonló módon be lehet látni, hogy $\mathbf{c_1} \in GRS_k(\mathbf{x}, \mathbf{v})$ és $\alpha \in \mathbb{F}$ esetén $\mathbf{c_2} = \alpha \mathbf{c_1}$ vektor is kódszó lesz. Ezen tulajdonságok miatt a $GRS_k(\mathbf{x}, \mathbf{v})$ kódszavai lineáris alteret alkotnak.

3.1.2. Tétel. *Az általános Reed-Solomon kódra ($GRS_k(\mathbf{x}, \mathbf{v})$) egyenlőséggel teljesül a Singleton-korlát, ezért MDS tulajdonságú.*

Vizsgáljuk meg két $\mathbf{c_1}, \mathbf{c_2} \in GRS_k(\mathbf{x}, \mathbf{v})$ Hamming-távolságát. Tegyük fel, hogy ez kevesebb, mint $n-k+1$. Ez azt jelent, hogy több, mint k helyen megegyeznek. Mit jelent ez az ezekhez a kódszavakhoz tartozó f_1, f_2 polinomokra nézve? Mivel legfeljebb $n-k$ ennek a két kódszónak távolsága, ezért $\mathbf{c_3} = \mathbf{c_1} - \mathbf{c_2}$ legalább k helyen lesz nulla, és a hozzá tartozó f_3 legfeljebb $(k-1)$-ed fokú polinom k különböző testelemen lesz nulla értékű. Ez csak úgy lehet, hogy ha az f_3 azonos a 0 polinommal. Ekkor viszont $\mathbf{c_1}$ és $\mathbf{c_2}$ azonosnak kell lennie, ami ellentmondás.

Tehát az $GRS_k(\mathbf{x}, \mathbf{v})$ kódban a minimális távolság $d \geq n-k+1$. Ugyanakkor a Singleton-korlát miatt tudjuk, hogy $d \leq n-k+1$, ezért $d = n-k+1$. Mivel egyenlőséggel teljesül a korlát, ezért az $GRS_k(\mathbf{x}, \mathbf{v})$ kód MDS tulajdonságú.

3.1.3. Megjegyzés. A bizonyítások alapján az is megkapható, hogy minden egyes $c \in GRS_k(\mathbf{x}, \mathbf{v})$ kódszóhoz pontosan egy legfeljebb $k-1$ fokú $f \in \mathbb{F}[x]$ polinom tartozik, mivel ilyen típusú polinomból ugyanannyi darab van, mint amennyi kódszót tartalmaz $GRS_k(\mathbf{x}, \mathbf{v})$, és egy polinomhoz legfeljebb egy kódszó tartozhat.

Minden $GRS_k(\mathbf{x}, \mathbf{v})$ kód MDS tulajdonságú, de vannak olyan MDS tulajdonságú kódok, amelyek nem írhatóak fel $GRS_k(\mathbf{x}, \mathbf{v})$-kódként, például a Hamming[7,4] kód. Ugyanakkor Ball bebizonyította a 2012-es cikkében[1], hogy prím elemszámú test fölött nem lehet a leghosszabb Reed-Solomon kódnál hosszabb MDS kódot létrehozni.

3.2. $\mathbf{u} \in \mathbb{F}^k$ üzenet kódolása

Két fő mód van egy $\mathbf{u} \in \mathbb{F}^k$ üzenet kódolására. Az egyik lehetőség az, hogy az üzenetet úgy értelmezzük, mint egy polinom együtthatói lennének. A másik lehetőség pedig az hogy, az üzenetben szereplő számokat úgy értelmezzük, mint egy polinom bizonyos helyeken felvett értékei. Az előző szakaszban megjegyeztük, hogy minden kódszóhoz pontosan egy legfeljebb $k-1$ fokú polinom tartozik, ezért a polinom meghatározása után már tudni fogjuk a kódszót.

3.2.1. Üzenet értelmezése, mint egy polinom együtthatói:

Vegyük a következő f polinomot:

$$f_u(x) = \sum_{i=1}^{k} u_i \cdot x^{i-1}$$

Az így kapott f polinomot értékeljük ki a $GRS_k(\mathbf{x}, \mathbf{v})$ kód $x_1, x_2, ..., x_n$ helyein, majd szorozzuk meg a $v_1, v_2, ..., v_n$ testelemekkel. Az így kapott kódszó:

$$\mathbf{w} = (v_1 f_u(x_1), v_2 f_u(x_2), ..., v_n f_u(x_n))$$

Vegyük észre, hogy az $\mathbf{w}$ vektor kiszámítása során az u_i testelemek mindegyike az első hatványon szerepel. Emiatt ezt a kódolást lehet egy lineáris transzformációként is kezelni, amely $GRS_k(\mathbf{x}, \mathbf{v})$ esetén a következő mátrix(ok szorzata):

$$G = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ x_1 & x_2 & x_3 & \cdots & x_n \\ & \vdots & & \ddots & \\ x_1^k & x_2^k & x_3^k & & x_n^k \end{pmatrix} \cdot \begin{pmatrix} v_1 & & & & \\ & v_2 & & & \\ & & v_3 & & \\ & & & \ddots & \\ & & & & v_n \end{pmatrix}$$

Az így kapott G mátrix generátormátrixa lesz a kódnak, és az üzenet kódolását úgy is értelmezhetjük, mint a generátormátrixszal való szorzást.

3.2.2. Üzenet értelmezése, mint egy polinom értékei:

Egy másik lehetőség az üzenet kódolására az, ha az üzenet egyes értékeit egy polinom bizonyos helyein felvett értékeiként kezeljük. Ebben az esetben a polinomot ki lehet számítani valamilyen polinominterpolációs algoritmus segítségével, például Lagrange interpolációval.

Legyen adva egy $GRS_k(\mathbf{x}, \mathbf{v})$ kód és egy $\mathbf{u} \in \mathbb{F}^k$ üzenet adva. a cél egy olyan f legfeljebb $(k-1)$ fokú polinom találása, amelyre $i \in [k]$ index esetén teljesül, hogy $f(x_i) = u_i$.

A Lagrange-interpolációhoz vezessük be a Lagrange-alappolinomokat:

$$l_i(x) = \prod_{\substack{1 \le j \le k \\ i \neq j}} \frac{x - x_j}{x_i - x_j}$$

Ezen polinomokra teljesülni fog, hogy $l_i(x_j) = 0$ ha $i \neq j$ és $l_i(x_i) = 1$ minden $i \in [k]$ indexre. Ezek segítségével már elő lehet állítani a feladatleírásnak megfelelő f polinomot az alappolinomok összegeként:

$$f(x) = \sum_{i=1}^{k} u_i \cdot l_i(x)$$

Ezt a polinomot a definíció szerint kiértékelve megkapjuk a keresett kódszót.

Habár ez a módszer elsőre összetettebbnek tűnik az előzőnél, számítógéppel gyorsan számolható. Vegyük észre, hogy ezzel a módszerrel számolva is egy lineáris transzformáció lesz az üzenet kódolása. Az ehhez tartozó mátrix elemeit megkaphatjuk egy n soros és k oszlopos G mátrixként, ahol az elemeket megkaphatjuk $G_{i,j} = v_j u_j l_j(x_i)$ formában. Mivel az $i \neq j$ esetén $l_j(x_i) = 0$ ha $i, j \in [k]$, ezért $\mathbf{v} = \mathbf{1}$ esetén ($\mathbf{v}$ a csupa 1 vektor) az első k oszlop és sor által alkotott részmátrix az identitás mátrix lesz. Ezért számítógépes kódolásnál a kódszó első k jeleként küldhetjük az $\mathbf{u}$ üzenetet úgy ahogy kaptuk, és csak az utolsó $n - k$ jelnél kell számításokat végezni.

3.3. Általánosított Reed-Solomon kódok duálisa

Egy $GRS_k(\mathbf{x}, \mathbf{v})$ kód egy ellenőrző mátrixának a meghatározásához vizsgáljuk meg a duális kód egy generátormátrixát.

3.3.1. Tétel. *Egy $GRS_k(\mathbf{x}, \mathbf{v})$ kód duálisa egy $GRS_{n-k}(\mathbf{x}, \mathbf{v}')$ kód*

Bizonyítás: Vegyük a $GRS_{n-1}(\mathbf{x}, \mathbf{v})$ kódot. Az erre a kódra merőleges altér egy 1 dimenziós, n távolságú MDS kód lesz, mivel MDS kód duálisa is MDS kód. Ha a duális altérből veszünk egy nem 0 v' vektort, akkor ennek a vektornak egyik koordinátája sem lesz 0, mivel n távolságra van a 0 vektortól. Jelölje $\circ$ a koordinátánkénti szorzást, $\cdot$ a skaláris szorzást. Ekkor egy $\mathbf{c} \in GRS_{n-1}(\mathbf{x}, \mathbf{v})$ kódra:

$$\mathbf{c} \cdot \mathbf{v}' = (\mathbf{c} \circ \mathbf{v}') \cdot \mathbf{1} = (\mathbf{v} \circ p_c(\mathbf{x}) \circ \mathbf{v}') \cdot \mathbf{1} = \mathbf{0}$$

ahol $\deg(p_c(\mathbf{x})) < n - 1$.

Most vegyünk alacsonyabb dimenzióval $GRS_k(\mathbf{x}, \mathbf{v})$, és a $GRS_{n-k}(\mathbf{x}, \mathbf{v}')$ kódokat. Legyen $\mathbf{d} \in GRS_k(\mathbf{x}, \mathbf{v})$ és $\mathbf{e} \in GRS_{n-k}(\mathbf{x}, \mathbf{v}')$. Ekkor:

$$\mathbf{d} \cdot \mathbf{e} = (\mathbf{d} \circ \mathbf{e}) \cdot \mathbf{1} = (\mathbf{v} \circ \mathbf{v}' \circ p_d(\mathbf{x}) \circ p_e(\mathbf{x})) \cdot 1 = 0$$

mivel $\deg(p_d \cdot p_e) \leq (k-1) \cdot (n-k-1) < n-1$.

És ezért tetszőleges k esetén $GRS_k(\mathbf{x}, \mathbf{v})$ duálisa $GRS_{n-k}(\mathbf{x}, \mathbf{v}')$ lesz.

Ez alapján a $GRS_k(\mathbf{x}, \mathbf{v})$ egy ellenőrző mátrixa:

$$H = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ x_1 & x_2 & x_3 & \cdots & x_n \\ & \vdots & & & \vdots \\ x_1^{k-1} & x_2^{k-1} & x_3^{k-1} & \cdots & x_n^{k-1} \end{pmatrix} \cdot \begin{pmatrix} v_1' & 0 & 0 & \cdots & 0 \\ 0 & v_2' & 0 & \cdots & 0 \\ 0 & 0 & v_3' & \cdots & 0 \\ & \vdots & & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & v_n' \end{pmatrix}$$

ahol $\mathbf{v}'$ vektor a $GRS_k(\mathbf{x}, \mathbf{v})$ kód duálisához, egy $GRS_{n-k}(\mathbf{x}, \mathbf{v}')$-hoz tartozó vektor.

4. fejezet

Reed-Solomon kódok dekódolása

4.1. ML dekódolás NP teljessége

Általánosságban a Reed-Solomon kódok maximum-likelihood dekódolását nem lehet minden esetben elvégezni polinomiális futási időben, ahogy ezt Guruswami és Vardy a cikkükben [7] belátták.

Pontosabban a következő eldöntési problémáról látták be, hogy NP-teljes:
Feladat: Reed-Solomon kódok Maximum-Likelihood dekódolása (MLD-RS)
Adott: egy m egész szám, egy $\mathcal{D} = \{x_1, x_2, ..., x_n\}$ halmaz, amely n különböző elemet tartalmaz a $\mathbb{F}_{2^m}$ testből, egy $k > 0$ egész szám, egy $\mathbf{y} \in \mathbb{F}_{2^m}^n$ célvektor, és egy $t > 0$ egész szám.
Kérdés: Van-e olyan $\mathbf{c} \in RS_k(\mathcal{D})$ kódszó, amelyre $d(\mathbf{c}, \mathbf{y}) \leq t$?

Ahol $d(\mathbf{c}, \mathbf{y})$ a Hamming-távolságot jelöli a két vektor között.

A cikkükben a három-dimenziós párosítás (TDM) feladatot visszavezették egy RS üzenet dekódolására. A TDM feladat egy ismert NP-teljes probléma, ezért ha az MLD-RS problémának is legalább ilyen nehéznek kell lennie.

4.2. Sudan algoritmusa RS kódok dekódolására

Habár tetszőleges t esetén a ML dekódolás NP teljes, bizonyos t paraméterek esetén lehet polinomiális időben is dekódolni ilyen kódokat.

Sudan 1996-os cikkében [10] leírt egy polinomiális idejű algoritmust, amellyel több, mint $d/2$ hibát is tud javítani d távolságú kódok esetében. Az algoritmus eredménye nem feltétlenül egy darab kódszó lesz, hanem egy lista a lehetséges kódszavakkal. Az algoritmus a következő lépésekből áll:

4.2.1. Algoritmus (Sudan)**.** Bemenet: Legyen adva n darab $(x_i, y_i) \in \mathbb{F}^2$ pár adva, amely közül legalább t darab illeszkedik egy legfeljebb d fokú f polinomra, vagyis $y_i = f(x_i)$ legalább t helyen. Ezután:

1. Legyen $m = \lceil d/2 \rceil - 1$ és $l = \lceil \sqrt{2(n+1)/d} \rceil - 1$
2. Megkeresünk egy $Q(x, y) : \mathbb{F}^2 \to \mathbb{F}$ polinomot, melyre teljesülnek az alábbi feltételek:
   - $p(z) = Q(z, z^d)$ polinomban z foka maximum $m + ld$
   - $Q(x_i, y_i) = 0$ minden $(x_i, y_i) \in H$ esetében.
   - Q nem az azonosan 0 polinom.
3. Ezután ezt a Q polinomot irreducibilis faktorokra bontjuk.
4. A faktorok közül kiválasztjuk az $(y - f(x))$ alakú f polinomokat, amelyek d fokkal rendelkeznek és legalább t értékre teljesül, hogy $f(x_i) = y_i$.

Az algoritmust tovább lehet javítani, ahogy Guruswami és Sudan egy cikkükben [6] leírják, amely még tágabb keretek között tud hibát javítani.

Ezt az algoritmust Julia [3] nyelven implementáltam a Nemo [5] és Flint [9] komputeralgebra könyvtárat felhasználva. A programkönyvtár tartalmaz véges testek feletti számolásához és polinomok faktorizálásához metódusokat. A program által lekódolt algoritmus:

1. Kiszámoljuk és eltároljuk a m és l számokat az input adatok alapján.
2. Előállítunk egy megfelelő $Q(x, y)$ polinomot egy egyenletrendszer Gauss-eliminációjával.

3. A kapott polinomot faktorizáljuk a programkönyvtárban lévő metódus segítségével.

4. Végül az irreducibilis faktorok közül kiválasztjuk azokat, amelyek $\alpha y + f(x)$ alakúak, ahol f egy legfeljebb d fokú egyváltozós polinom.

Ezen felül még néhány lépést kell tenni, hogy az adatokat a programkönyvtár által is feldolgozható formátumban legyenek.

4.2.1. Az algoritmus helyességének a bizonyítása

4.2.2. Lemma. *Ha létezik egy ilyen tulajdonságokkal rendelkező Q polinom, akkor az megtalálható poly(n) futási időben.*

Bizonyítás: Vizsgáljuk meg a Q polinomot monomok összegeként. Ekkor a polinomot $Q(x,y) = \sum_{j=0}^{l} \sum_{k=0}^{m+(l-j)d} q_{k,j} x^k y^j$ alakban lehet írni, ahol $q_{k,j}$ a polinomnak az $x^k y^j$ monomhoz tartozó együtthatója. Ezután az $\forall (x_i, y_i) \in H : Q(x_i, y_i) = 0$ feltétel alapján fel lehet írni a következő egyenletrendszert, amelyekben $q_{k,j}$ lesznek az ismeretlen változók:

$$\forall i \in [n] : \qquad \sum_{j=0}^{l} \sum_{k=0}^{m+(l-j)d} q_{k,j} x_i^k y_i^j = 0$$

Ez egy homogén egyenletrendszer lesz, amelynek keressük egy nem azonosan 0 megoldását. Egy ilyen megoldást (ha van) ki lehet számolni poly(n) időben, például Gauss-eliminációval.

4.2.3. Lemma. *Ha $(m+1)(l+1) + d\binom{l+1}{2}/2 > n$, akkor létezik ilyen tulajdonságokkal rendelkező $Q(x,y)$ polinom, amely nem azonosan 0.*

Bizonyítás: A fokszámra vonatkozó megkötés miatt egy $x^k y^j$ monomban az x és y fokára teljesül, hogy $k + d \cdot j \leq m + ld$. Ezért k és j összesen $(m+1)(l+1) + d\binom{l+1}{2}/2$ különböző értéket vehet fel, tehát $Q(x,y)$ legfeljebb $(m+1)(l+1) + d\binom{l+1}{2}/2$ monomból áll valamilyen együtthatókkal.

Mivel az egyenletek és ismeretlenek számára teljesül, hogy $(m+1)(l+1)+d\binom{l+1}{2}/2 > n$, és mivel az egyenletrendszer homogén, vagyis az azonosan 0 ismeretlenek megoldják, ezért létezik másik megoldás is, ahol nem minden ismeretlen nulla.

4.2.4. Lemma. *Amennyiben létezik a feltételeknek megfelelő $Q(x, y)$ polinom, és $f(x)$-re teljesül, hogy $t > m + ld$ darab i indexre $f(x_i) = y_i$, akkor az $(y - f(x))$ polinom osztja $Q(x, y)$-t.*

Bizonyítás: Vizsgáljuk meg a $p(x) = Q(x, f(x))$ polinomot. Mivel $Q(z, z^d)$ polinomban z foka legfeljebb $m + ld$, az $f(x)$ polinomban x foka legfeljebb d, ezért $p(x)$ foka is legfeljebb $m + ld$ lesz.

Ez $p(x)$ polinom csak az azonosan 0 polinom lehet. Legalább t darab $(x_i, y_i) \in H$ párra teljesül, hogy $p(x_i) = Q(x_i, f(x_i)) = Q(x_i, y_i)$, ezért a $p(x)$ polinomnak legalább t darab különböző gyöke van, (mivel $x_i \neq x_j$ ha $i \neq j$). Mivel t nagyobb, mint a polinom foka, ezért ez csak úgy lehet, ha $p(x)$ azonosan 0-val.

Ezután tekintsük $Q(x, y)$ ne egy kétváltozós polinomnak $\mathbb{F}$ véges test felett, hanem egy egyváltozós $Q_x(y)$ polinomnak $\mathbb{F}[x]$ polinomgyűrű felett. Ha valamilyen $\zeta \in \mathbb{F}[x]$ értékre $Q_x(\zeta) = 0$, akkor ezt a gyököt ki lehet emelni, és ezért $(y - \zeta)$ osztja a $Q_x(y)$ polinomot. Az $\zeta = f(x)$ behelyettesítéssel megkapjuk a lemmát.

4.2.5. Lemma. *Az algoritmus akkor fog adott n mellett a lehető legtöbb t paraméterre sikerrel lefutni, ha $m = \lceil d/2 \rceil - 1$ és $l = \lceil \sqrt{(2(n+1))/d} \rceil - 1$.*

Bizonyítás: Az előző lemmák alapján két feltételt kaptunk az m és l paraméterekre. Egyfelől az $(y - f(x))$ polinom oszthatósága miatt:

$$m + ld < t$$

Másfelől a $Q(x, y)$ polinom létezése miatt:

$$(m + 1)(l + 1) + d\binom{l+1}{2} > n$$

A fenti egyenletet átrendezve azt kapjuk, hogy:

$$m > \frac{n + 1 - d\binom{l+1}{2}}{l + 1} - 1$$

Ezt az első egyenlőtlenségbe visszahelyettesítve kapunk egy korlátot becslést t-re:

$$t > m + ld > \frac{n+1-d\binom{l+1}{2}}{l+1} - 1 + ld + 1 = \frac{n+1}{l+1} + \frac{dl}{2}$$

A fenti egyenlőtlenség jobb oldalán lévő kifejezést szeretnénk minimalizálni az l paraméter megfelelő választásával. A kifejezés akkor veszi fel a szélső értékét, ha l-re a következő teljesül:

$$\frac{-(n+1)}{(l+1)^2} + \frac{d}{2} = 0 \Rightarrow l = \sqrt{\frac{2(n+1)}{d}} - 1$$

Amely alapján m értékére teljesül, hogy:

$$m \geq \sqrt{\frac{(n+1)d}{2}} - \sqrt{\frac{(n+1)d}{2}} + \frac{d}{2} - 1 = \frac{d}{2} - 1$$

És ekkor t-re:

$$t > m + ld \geq \frac{d}{2} - 1 + \left(\sqrt{\frac{2(n+1)}{d}} - 1\right) d = \sqrt{2(n+1)d} - \frac{d}{2} - 1$$

Mivel az m és l paramétereknek egész számoknak kell lenniük, ezért ezek az egyenlőtlenségek nem mindig teljesülnek egyenlőséggel.

Ezek alapján ki lehet mondani a következő tételt:

4.2.6. Tétel. *Legyen adva n különböző (x_i, y_i) pár, ahol a pár tagjai az elemei valamilyen $\mathbb{F}$ testnek. Legyen adva továbbá két egész szám, t és d, hogy teljesüljön az $t \geq \lceil\sqrt{2(n+1)d}\rceil - \lfloor d/2\rfloor - 1$ egyenlőtlenség.*

Ekkor létezik egy olyan algoritmus poly(n) futási idővel, amely megtalálja az összes olyan legfeljebb d fokú f polinomot, amelyre legalább t index esetén teljesül az $y_i = f(x_i)$ egyenlőség.

A. függelék

Számolás véges testek elemeivel

A bizonyítások során a műveleteket általában valamilyen $\mathbb{F}_q$ véges test fölött végeztük. Ugyanakkor a legtöbb mai számítógép nem tartalmaz parancsokat véges testelemek összeadására vagy szorzására, hanem csak kis egész számok összeadására, szorzására és maradékos osztására használható parancsokat tartalmaz. A mai számítógép alatt az x86_64 architektúrájú processzorral működő gépeket értem.

A.1. Általános ($|\mathbb{F}_q| = p^n$) eset

Kis p és $n = 1$ esetén az $\mathbb{F}_p$ esetet egyszerűen lehet kezelni. A testelemeket $0 \leq x \leq p-1$ közötti egész számokként tároljuk. A testbeli összeadást és a szorzást a és b között elvégezhetjük úgy, hogy először a kis egész számok körében összeadjuk vagy szorozzuk a és b számokat, majd vesszük a kapott

Ha $n \geq 2$ esetén két megközelítés szoktak alkalmazni véges testelemek szorzására.

Az egyik lehetőség, hogy minden testelemhez hozzárendelünk egy egész számot, és valamilyen módon eltároljuk a memóriában a test additív és multiplikatív szerkezetét. Ugyanakkor csak kis testek esetén lehet ilyen megoldást alkalmazni, ugyanis egy összeadó és szorzótábla tárolásához a testelemek számával négyzetesen arányos mennyiségű memóriát kell használni.

A gyakrabban használt másik lehetőség, amelyet az általam használt Flint prog-

ramkönyvtár is használ, hogy a testelemeket $q = p^n$ esetén egy $n-1$ fokú $\mathbb{F}_p$ feletti polinomként kezeli, a polinom együtthatóit egy tömbben tárolva.

Két $a, b \in \mathbb{F}_q$ testelem összeadást elemenként el lehet végezni együtthatóként $\mathbb{F}_p$ felett, majd az eredményként kapott eltároljuk egy $c \in \mathbb{F}_q$ testelemként. A reprezentációknál teljesülni fog, hogy $c_i = a_i + b_i$ minden i index esetén, ahol $a_i, b_i, c_i \in \mathbb{F}_p$ az alaptest elemei, és az x^i monomhoz tartozó együtthatókat jelölik a megfelelő polinomban.

Két $a, b \in \mathbb{F}_q$ elem szorzását úgy valósíthatjuk meg, hogy a hozzájuk tartozó polinomokat összeszorozzuk, és a kapott szorzatpolinomot maradékosan elosztjuk egy előre kiválasztott n fokú primitív irreducibilis polinommal. Az osztás maradékaként kapott polinom lesz a test feletti szorzás eredménye.

Mivel az azonos elemszámú testek egymással izomorfak, ezért irreducibilis polinom választása tetszőleges lehet. A programkönyvtárakban azonban általában a $C_{p,n}$ Conway polinomot szokták használni konvencióként.

A.1.1. Definíció (Conway polinom)**.** A $C_{p,n}$ Conway polinom az a (lexikografikusan legkisebb) irreducibilis n fokú polinom $\mathbb{F}_p[x]$ fölött, amelyre teljesülnek a következő feltételek:

1. $C_{p,n}$ főegyütthatója 1.

2. $C_{p,n}$ primitív, vagyis minden gyöke primitív elem a polinom által generált testben.

3. Minden $m|n$ esetén $C_{p,m}\left(x^{(p^n-1)/(p^m-1)}\right) \equiv 0 \pmod{C_{p,n}}$, vagyis $C_{p,n}$ minden gyökének az $\frac{p^n-1}{p^m-1}$-edik hatványa az $C_{p,m}$ polinom egy gyöke.

A.1.2. Definíció ($\mathbb{F}_p$ feletti d fokú polinomok lexikografikus rendezése)**.** Feleltessük meg az $\mathbb{F}_p$ elemeinek az $0, 1, ..., p-1$ számokat. Ez alapján az $\mathbb{F}_p$ elemeit rendezhetjük $0 < 1 < ... < p-1$ szerint.

Legyen adva egy f és egy g polinom, $f(x) = \sum_{i=0}^{d}(-1)^i a_i x^{d-i}$ és $g(x) = \sum_{j=0}^{d}(-1)^j a_j x^{d-j}$ alakban. Az f polinom lexikografikusan nagyobb, mint g, ha $i < n$ esetén $a_i = b_i$, és $a_n > b_n$.

A lexikografikus rendezés helyett másfajta rendezést is használhatnánk. Ezen megkötés miatt teljesül, hogy minden p és n esetén pontosan egy ilyen polinom van, és emiatt erre egyfajta konvencióként lehet tekinteni, amely segítségével a különböző programok között ugyanolyan reprezentációt lehet használni a véges testelemek jelzésére.

Egy a testelem multiplikatív inverzének a kiszámításához fel lehet azt a tényt, hogy a véges testek multiplikatív csoportja egy ciklikus csoport $p^m - 1$ elemmel, és emiatt fenn áll az $a^{-1} = a^{p^m-1}$ egyenlőség. Az a^{p^m-1} elemet ki lehet számolni ismételt hatványozással $\lceil \log_2(p^m-1) \rceil \leq \lceil \log_2(p) \cdot m \rceil$ szorzási művelet segítségével.

Az inverz elem ismeretében lehet a testelemeket osztani is egymással $a/b = a \cdot (b^{-1}) = a \cdot b^{p^m-1}$ átírással.

Az általános véges testek feletti szorzást a Nemo/Flint programkönyvtárral végeztem el. Az alábbi programkóddal egy nemnegatív egész számot átváltok egy testelemre, úgy, hogy a számnak a test karakterisztikája szerinti számrendszerben vett számjegyei lesznek a testelemhez tartozó polinom együtthatói. Az átváltásra a következő programot írtam:

```
function int_to_felem(n, 𝔽 ::FinField)
    @boundscheck 0 ≤ n < order(𝔽) # $n is out of range for $𝔽
    g, c, f0, f1 = gen(𝔽), characteristic(𝔽), zero(𝔽), one(𝔽)
    f = f0
    exp = f1
    while n > 0
        f += 𝔽(n % c) * exp
        n ÷= c
        exp *= g
    end
    return f
end
→(n ::I, 𝔽 ::FinField) where I <: Integer = int_to_felem(n, 𝔽)
→(n ::AbstractArray, 𝔽 ::FinField) where I <: Integer = int_to_felem.(n, (𝔽,))

function felem_to_int(f)
    𝔽 = parent(f)
    g = gen(𝔽),
    c = UInt(characteristic(𝔽))
    exp = 1
    n = 0
```

```
    while f != 𝔽(0)
        n += coeff(f, 0) * exp # coeff(f, 0) is the coeff of f^0 in the repr
        f = 𝔽(f - coeff(f, 0)) // g
        exp *= c
    end
    return n
end
→(f ::Union{F, AbstractVector{F}}, ::Type{I}) where
    {F <: FinFieldElem, I <: Integer} =
    I.(felem_to_int.(f))

function →(f ::F, 𝔽 ::FinField) where {F <: FinFieldElem}
    @assert parent(f) == 𝔽 "Cannot convert $f into $𝔽"
    return f
end
→(f ::AbstractVector{F}, 𝔽 ::FinField) where {F <: FinFieldElem} = f .→ (𝔽,)

# backport
if VERSION < v"1.6.1"
    const var"'ᵀ" = Base.transpose
end
```

Ezek a módszerek működnek 2 karakterisztikájú testek esetén is, de, mint ahogy az általában lenni szokott, lehet ezeket a testeket speciális esetként kezelni.

A.2. Számolás a 2 karakterisztikájú testelemekkel

A legtöbb mai processzor 2-es számrendszerben végez számításokat, és ezért bizonyos, a processzorba beépített parancsok segítségével $\mathbb{F}_{2^n}$ feletti számításokat fel lehet gyorsítani.

Egyrészről az $\mathbb{F}_2$ feletti polinomokat nem kell jegyenként külön tárolni egy tömbben, hanem tárolhatjuk egy kettes számrendszerbeli számként, ahol a 2^k helyi értékhez tartozó jegy megegyezik az x^k együtthatójával. Például a $x^4 + x + 1$ polinomot átváltjuk a $10011_2 = 19_{10}$ számnak, és ezt a számot tároljuk a memóriában. Ezáltal tömörebben tárolhatjuk el az együtthatókat, és így kevesebb helyet foglal el egy testelem a számítógép memóriájában.

Másrészről a processzornak vannak olyan beépített utasításai számok kezelésére, amelyet kihasználhatunk ebben a reprezentációban. Két polinom összeadását

ebben a formában elvégezhetjük a XORQ művelet segítségével. A XOR műveletet logikai kapuk szintén a következő módon lehet értelmezni: Van két bemeneti bit. Ha két bit különbözik, akkor igaz/1 értéket ad vissza, ha megegyezik, akkor hamis/0 értéket ad vissza. Vegyük észre, hogy ez megegyezik az $\mathbb{F}_2$-beli összeadással. A XORQ ezt a művelet alkalmazza két 64-bites szám esetében a következő módon: Veszi a két szám 2-es számrendszeri alakját, majd a helyi értékeken sorra menve alkalmazza a XOR műveletet. Mivel a testelemeket együtthatóit egy szám 2-es számrendszerbeli jegyein tároltuk, a XORQ művelet megfelel két testelemek összeadásának.

A testelemek szorzásának a gyorsítására két lehetőség van. Az egyik lehetőség a GFNI műveletcsomag használata, de ezzel csak a $\mathbb{F}_{2^8}$ test elemeit lehet szorozni, és csak egyes a 2019 után készült processzorok támogatják ezeket az utasításokat. A másik lehetőség a CLMUL műveletcsomag segítségével lehet, amelyet majdnem minden 2010 után készült processzor támogat, és amely csomagból használtam műveleteket a $\mathbb{F}_{2^{64}}$ fölötti testelemek szorzására.

A (V)PCMULQDQ művelet segítségével két 64-bites számot "átvitel-nélkül" szorozhatunk össze. A számokat értelmezzük két (legfeljebb) 63-ad fokú polinomként $\mathbb{F}_2$ felett, ahol a biteket megfeleltetjük az együtthatóknak. Ezt a két polinomot szorozzuk össze. Eredményként két 64-bites számot kapunk, az egyik a szorzatpolinom 0-tól és 63-ig, a másik a szorzat 64-től 127-ig terjedő együtthatóit tárolja.

Tároljuk a $\mathbb{F}_{2^{64}}$ elemeit polinomként, a polinomok együtthatóit 64-bites számokként, és a szorzás során egyszerűsítsünk az $c_{2,64} = x^{64} + x^{33} + x^{30} + x^{26} + x^{25} + \\ + x^{24} + x^{23} + x^{22} + x^{21} + x^{20} + x^{18} + x^{13} + x^{12} + x^{11} + x^{10} + x^7 + x^5 + x^4 + x^2 + x + 1$ polinommal. Vezessük be még a $d = c_{2,64} - x^{64}$ polinomot. A testbeli szorzást három átvitel-nélküli szorzással, egy sima szorzással és három bitenkénti kizáró-vagy művelettel el lehet végezni.

Vegyünk két legfeljebb 63 fokú a és b polinomot, és szorozzuk össze őket. A kapott eredmény egy legfeljebb 126 fokú r_1 polinom, amelyet két részre van bontva $r_1 = r_{1,high}x^{64} + r_{1,low}$. Ennek a polinomot $c_{2,64}$-es polinommal vett maradéka kell. Szorozzuk össze $r_{1,high}$-t és d-t, így kapjuk az $r_2 = r_{2,high}x^{64} + r_{2,low}$ polinomot, és vizsgáljuk az $r_{1,low} + r_2$ polinomot. Mivel d foka 33, ezért ennek a polinomnak

nincs $126 - 64 + 33 = 95$-nél magasabb fokú tagja. Ugyanakkor a $c_{2,64}$-gyel vett maradéka nem változott.

Ezt a lépést még egyszer ismételve kapjuk a legfeljebb $95 - 64 + 33 = 64$ fokú $r_{1,low} + r_{2,low} + r_3$ polinomot, ekkor $r_{3,high}$ vagy 0 vagy 1. Mivel a sima szorzás gyorsabb művelet, minta az átvitel-nélküli szorzás, vagy az elágazás, ezért az utolsó lépésben $r_{4,low} = r_{3,high} * d$ lesz. A szorzások után három összeadás kell, $r_{1,low} + r_{2,low} + r_{3,low} + r_{4,low}$, és ez lesz a testelemek feletti szorzás eredménye.

A testművelete leíró GL2P64 programkód:

```
"""
Implements arithmetric operations for the GF(2^64) field.
"""
struct GF2P64 <: Number
    value ::UInt64
end

const _m128 = NTuple{2,VecElement{UInt64}}
    function Base.:*(a ::GF2P64, b ::GF2P64)
    vd = _m128(( 0x0000000247f43cb7, UInt64(0) )) # Conway(2, 64)
    va = _m128(( a.value, 0 )) # a
    vb = _m128(( b.value, 0 )) # b
    vr1 = ccall("llvm.x86.pclmulqdq", llvmcall, _m128, (_m128, _m128, UInt8),
    va, vb, 0x00) # a * b
    vr2 = ccall("llvm.x86.pclmulqdq", llvmcall, _m128, (_m128, _m128, UInt8),
    vr1, vd, 0x01) # vr1[high] * Conway(2,64)
    vr3 = ccall("llvm.x86.pclmulqdq", llvmcall, _m128, (_m128, _m128, UInt8),
    vr2, vd, 0x01) # vr2[high] * Conway(2,64)
    vr4 = _m128(( vr3[2].value * vd[1].value, 0 )) # vr3[2] is 0 or 1
    return GF2P64(xor(vr1[1].value, vr2[1].value, vr3[1].value, vr4[1].value))
end

Base.:+(a ::GF2P64, b ::GF2P64) = GF2P64(a.value ⊻ b.value)
Base.:-(a ::GF2P64, b ::GF2P64) = GF2P64(a.value ⊻ b.value)
Base.:+(a ::GF2P64) = a
Base.:-(a ::GF2P64) = a

# 2^64 - 2 as a bit pattern: 63 ones and a zero at the 2^0 place
function Base.inv(a ::GF2P64) # a^-1 = a^(2^64-2)
    @assert a.value != UInt64(0)
    val = one(a)
    a *= a
    for _ in 2:64
```

```
        val *= a
        a *= a
    end
    return val
end
Base.://(a ::GF2P64, b ::GF2P64) = a * inv(b)

function Base.:^(a ::GF2P64, p ::Integer)
    if p < 0 return inv(a)^(-p) end
    val = one(a)
    while p > 0
        if p % 2 == 1
            val *= a
        end
        a *= a
        p >>= 1
    end
    return val
end

Base.zero(::GF2P64) = GF2P64(0)
Base.zero(::Type{GF2P64}) = GF2P64(0)
Base.one(::GF2P64) = GF2P64(1)
Base.one(::Type{GF2P64}) = GF2P64(1)
function Base.rand(rng ::AbstractRNG, ::Random.SamplerType{GF2P64})
    GF2P64(rand(rng, UInt64))
end
Base.parent(::GF2P64) = GF2P64

→(x, ::Type{GF2P64}) = GF2P64.(x)
UInt64(x ::GF2P64) = x.value
→(x ::AbstractArray{GF2P64}, ::Type{UInt64}) = UInt64.(x)
```

B. függelék

Üzenetek kódolása $GRS_k(\mathbf{x}, \mathbf{v})$ kóddal

Mielőtt rátérnénk az üzenetek kódolására, definiáljunk a $GRS_k(\mathbf{x}, \mathbf{v})$ kódok kezelésére egy osztályt:

```
struct GRS
    k
    x
    v
    𝔽
    GRS(k, x, v) = new(k, x, v, parent(x[1]))
end
RS(k, x) = GRS(k, x, fill(one(x[1]), size(x)))
```

A 3.2 szakaszban bemutattam két lehetséges módot az üzentek elkódolására. Ebben a szakaszban leírom a részletesebb implementációját a kódolási módszernek. Az üzeneteket egy generátormátrixszal szorzás eredményeként kódolom, ezért a generátormátrixokat érdemesebb alaposabban megvizsgálni. A példák bemutatásához használjuk a $\mathbb{F}_{257}$ test felett az $\mathbf{x} = (1, 2, 3, 4, 5, 6, 7)$ vektorral az RS(4, $\mathbf{x}$) kódot. A programnyelv egydimenziós vektorok közül csak az oszlopvektorokat támogatja, ezért mátrixszorzás előtt és után transzponálni kell a vektorokat.

Az első módszernél egy polinomot kell kiértékelni az együtthatói alapján. Te-

gyük fel, hogy a polinom együtthatóit egy sorvektorban tároljuk, és a kiértékelendő helyeket az $\mathbf{x}$ tartalmazza. A polinom kiértékelése ugyanaz, mintha egy Vandermonde mátrixszal szoroznánk, amit az $\mathbf{x}$ értékei alapján állítottunk össze. A példa kódhoz tartozó generátormátrix:

$$G = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 1 & 4 & 9 & 16 & 25 & 36 & 49 \\ 1 & 8 & 27 & 64 & 125 & 216 & 86 \end{pmatrix}$$

És a mátrix generálásához tartozó programkód:

```
vandermonde(x, height) = [
    xᵢ^j for # value
    j in 0:height-1, # row
    xᵢ in x # column
]

generator_coeff(code ::GRS) = vandermonde(code.x, code.k) .* code.v'ᵀ

encode_coeff(u, code ::GRS) = (u'ᵀ * generator_coeff(code))'ᵀ
```

A második esetben az üzentet egy polinom felvett helyein értelmeztük. Ekkor a kódolást két lépésben végezzük: először keresünk egy megfelelő polinomot, majd ezt a polinomot kiértékeljük a megfelelő helyeken. A kiértékelést ugyanúgy lehet végezni, mint az előző esetben. A megfelelő polinomot Lagrange-interpoláció segítségével lehet megtalálni.

Legyenek $z_1, z_2, ..., z_k$ a kiértékelési helyeket. Ezeket megválaszthatjuk a $\mathbf{x}$ vektor első k helyen lévő értékeinek. Készítsünk egy L mátrixot a következő módon: A mátrix i-edik sorában legyenek a z_i ponthoz tartozó l_i interpolációs együttható, az együtthatók balról jobbra haladva legyenek a növekvő kitevők szerint, vagyis az első oszlopban legyen a konstans együttható, az utolsó oszlopban pedig a főegyüttható. Ha ezt L mátrixot megszorozzuk az u üzenethez tartozó sorvektorral, akkor keletkező sorvektor együtthatóit egy polinomnak megfeleltetve éppen az interpoláció eredményét kapjuk meg.

Az előző művelet után kapott polinomot pedig kiértékelhetjük eg V Vandermonde mátrix segítségével. A generátormátrix ennek a kettőnek a szorzata lesz, mivel a mátrixszorzás asszociatív.

$$L = \begin{pmatrix} 4 & 167 & 130 & 214 \\ 251 & 138 & 253 & 129 \\ 4 & 250 & 132 & 128 \\ 256 & 216 & 256 & 43 \end{pmatrix}$$

$$G = L \cdot V = \begin{pmatrix} 1 & 0 & 0 & 0 & 256 & 253 & 247 \\ 0 & 1 & 0 & 0 & 4 & 15 & 36 \\ 0 & 0 & 1 & 0 & 251 & 237 & 212 \\ 0 & 0 & 0 & 1 & 4 & 10 & 20 \end{pmatrix}$$

Az ehhez a módszerhez tartozó programkód:

```
function lbase(x) # coeffs of the base polynomials in lagrange interpolation
    f0, f1 = zero(x[1]), one(x[1])
    n = length(x)
    # initialize with constant one polynomials
    lbase = [c == 0 ? f1 : f0 for r in 1:n, c in 0:n-1]
    # l_i = ... // (x[i] - x[j])
    for i in 1:n, j in 1:n
        if i == j continue end
        lbase[i, begin] //= x[i] - x[j]
    end
    for j in 1:n, i in 1:n
        if i == j continue end
        # f2 = t*f1 + ai * f1
        for k in j + (i > j):-1:2
            lbase[i, k] = (lbase[i, k-1] - x[j] * lbase[i, k])
        end
        lbase[i, begin] *= - x[j] # constant coeff
    end
    return lbase
end

# The first k column will be identity, that calculation can be skipped
generator_inpol(code ::GRS) = hcat(I, lbase(code.x[1:code.k]) *
    vandermonde(code.x[code.k+1:end], code.k)) .* code.v'^T

encode_inpol(u, code ::GRS) = (u'^T * generator_inpol(code))'^T
```

A GF2P64 formátumban az $\mathbb{F}_{2^64}$ véges test fölött a program gyorsan működik. Egy $n = 120$, $k = 40$ paraméter segítségével egy 100 megabájtos üzenetet (40×8 bájtos blokkokra osztva) $2,6$ másodperc alatt kódol el az interpolációs módszerrel egy 2017-es laptopon. A tömbösített futtatáshoz az alábbi metódust írtam:

```
function encode_data(data ::Vector{I}, grs ::GRS) where I <: Integer
    @boundscheck length(data) % grs.k == 0
    data = data → grs.𝔽
    f0, f1 = zero(grs.𝔽), one(grs.𝔽)
    @assert all(grs.v .== f1)
    n = length(data) ÷ grs.k
    k = grs.k
    l = length(grs.x) - k
    ret = Vector{typeof(f0)}(undef, l*n)
    gok = generator_inpol(grs)[:, grs.k+1:end] # cut the identity
    for i in 1:n
        block = data[(i-1)*k+1 : i*grs.k]
        ret[(i-1)*l+1 : i*l] = (block'^T * gok)'^T
    end
    return ret → I
end
```

C. függelék

$GRS_k(\mathbf{x}, \mathbf{v})$-ből származó hibás üzenetek dekódolása Sudan algoritmusával

Ebben a fejezetben részletesebben leírom az algoritmus implementációját.

C.1. Kezdőértékek beállítása

A programhívásban lehetőség van arra, hogy egy kapott üzentet, és a kódolásnál használt $GRS_k(\mathbf{x}, \mathbf{v})$ kód paramétereit adjuk meg bemenetként. Ekkor a bemenethez használt $\mathbf{x}$ vektort a kódból veszi, a legmagasabb fok $d = k - 1$ lehet, az üzenetet koordinátánként elosztja a $\mathbf{v}$ értékeivel, mert az algoritmusban nem GRS, hanem RS kódokat használnak, végül pedig a véges testet meghatározza a kód alapján.

Az kód hosszát nem kell direkt paraméterként átadni, mivel bemeneti vektorok hosszát ki lehet számolni. Tároljuk el ezt n-ben.

Az cikkben leírtak szerint beállítjuk az m és l értékeket.

Végül hozzunk létre egy objektumot a test fölött kétváltozós polinomgyűrűk kezeléséhez. A polinomgyűrű-t jelölje az R betű, a két változó legyen a (nem megvastagított) x és y betű.

Az ehhez tartozó programkód:

```
function sudan_decoder(y, grs ::GRS)
    y .//= grs.v
    return sudan_decoder(grs.x, y, grs.k-1, grs.𝔽)
end

function sudan_decoder(x, y, d, 𝔽)
    @boundscheck length(x) == length(y)
    @assert all(parent.(x) .== parent.(y) .== (𝔽,))
    n = length(x)
    m = ceil(Int, d/2) - 1
    l = ceil(Int, √(2(n+1)/d)) - 1
    R, (x, y) = 𝔽['x', 'y'] # make polynomial ring over F
    #...
```

C.2. $Q(x, y)$ polinom kiszámítása

Sudan a cikkében leírt egy algoritmust $Q(x, y)$ polinom kiszámítására egy egyenletrendszer segítségével. Az általam használt programcsomag azonban nem tartalmaz olyan függvényt, amely ezt a részfeladatot direktbe megoldaná, ezért a következő algoritmus implementáltam:

C.2.1. Algoritmus ($Q(x, y)$ meghatározása programmal)**.** Első lépésben legeneráljuk az összes lehetséges $x^k y^j$ monomot, és ezeket eltároljuk egy vektorba. Minden $x^k y^j$ monomhoz ez alapján a vektor alapján hozzá lehet rendelni egy i' sorszámot.

Ezután létrehozunk egy M mátrixot, melynek oszlopait a monomok határozzák meg, és sorait a bemenetként kapott (x_i, y_i) párok. Az i-edik sorhoz és i' oszlophoz tartozó elem legyen $M_{i,i'} = monom_{i'}(x_i, y_i)$, ahol $monom_{i'}$ az i'-es sorszámú $x^k y^j$ monom.

A mátrixot értelmezhetjük egy homogén egyenletrendszerként, aminek keressünk egy nem azonosan 0 megoldását.

Egy ilyen megoldást lehet találni számítógép segítségével egy speciális Gauss-eliminációval. A Gauss-elimináció során az oszlopokon jobbról balra haladva kere-

sünk elemeket pivotáláshoz a még nem kiválasztott sorokból, majd a talált pivotált elem sorát kiválasztjuk, és a valahányszorosát kivonjuk a maradék sorokból úgy, hogy a pivot elem oszlopában csak 0 legyen a maradék sorokban. A végén a sorok átrendezésével kapunk egy "lépcsős" alakú mátrixot, ahol lépcső fokain a pivot elemnek találhatóak.

A Gauss-elimináció után az ismeretlen változókat fel lehet osztani pivot/kötött változókra, (amelyek oszlopaiban pivotálást végeztünk), és free/szabad változókra, (amelyek oszlopaiban nincs pivot elem). A szabad változókat akárhogyan megválaszthatjuk, ezért beállíthatjuk 1-nek is. A kötött változókat ezután egyszerűen ki lehet számolni, a sorokon visszafele haladva és a szabad változókat behelyettesítve. Mivel több ismeretlen van, mint egyenlet, ezért lesznek szabad változók, és ezért az elimináció után egy nem mindenhol 0 megoldást számol ki a program.

A következő lépésben a kapott megoldás alapján a beírjuk az együtthatókat a polinomba. Az egyenletrendszer megoldását értelmezhetjük vektorként, amelyet jelöljünk *megold*-nak. Ebben a vektorban ugyanannyi elemet tartalmaz, mint ahány oszlopa van az M mátrixnak. A Gauss-elimináció során az oszlopok sorrendjében nem történt változás, ezért az i' sorszámú elem a megoldásvektorban megfelel az i' sorszámú monom együtthatójának.

Tehát a $Q(x, y)$ polinomot megkaphatjuk $Q(x, y) = \sum megold_{i'} \cdot monom_{i'}$ alakban.

Az algoritmus lefuttatásához $O(n\,(ml + dl^2)\,(n + log(m) + log(l)))$-mel arányos darab műveletet kell elvégezni a testelemek között. Az elimináció során legfeljebb n sorban történik pivotálás, és minden pivotálás során sorszámszor, n-szer oszlopszámmal, $((m + 1)(l + 1) + d\binom{l+1}{2}/2)$-val arányos mennyiségű műveletet kell elvégezni, és ez összesen $O(n^2\,(ml + dl^2))$ testművelet. A visszahelyettesítés során minden sorban legfeljebb oszlopszámnyi tesműveletet kell elvégezni, amely $O(n\,(ml + dl^2))$-rel arányos mennyiségű testművelet, de az elimináció során egy n nagyságrenddel több műveletet kell elvégezni, ezért nem ez lesz a befolyásoló tényező. A mátrix generálása során minden elemhez $x^k y^j$ monomot ki kell értékelni, és ehhez $O(n\,(ml + dl^2)\,(log(m) + log(l)))$-vel arányos mennyiségű testművelet kell.

Az algoritmushoz tartozó programkódok:

```
#...
# store every possible monom into a list
monoms = [x^k * y^j for j in 0:l for k in 0:m+(l-j)d]
M = [ # equation system as a matrix
    monom(x_i, y_i) for # evaluate the monom with the pair
    (x_i, y_i) in zip(x, y), # (x_i, y_i) pairs as row indexes
    monom    in monoms     # monoms as columns indexes
]
coeffs = gauss_solve!(M) # get the coefficients as the solution
# Q is the sum of the monomials with the corresponding coefficients
Q = sum(coeffs .* monoms)
#...
```

És a meghívott függvény:

```
"""
Returns an x vector satisfying the M*x .== 0 equations.
If there are multiple solutions, x will not be everywhere zero.
"""
function gauss_solve!(M ::AbstractMatrix)
    Nrow, Ncol = size(M) # number of rows and columns
    f0 = zero(M[1, 1]) # zero element from the field
    f1 = one(M[1, 1]) # one element from the field
    pivot = [0] # insert an extra initial zero
    for prow in 1:Nrow # maximal number of pivot rounds
        for c in last(pivot)+1:Ncol, r in prow:Nrow
            if M[r, c] != f0 # search for non-zero element in the matrix
                push!(pivot, c) # pivot at the c column
                if prow != r
                    M[[prow, r], :] = M[[r, prow], :] # swap rows
                end
                pinv = inv(M[prow, c]) # inverse of the pivot element
                M[prow, :] .*= pinv # normalize the pivot row
                # subtract pivot row
                @. M[prow+1:Nrow, :] -= M[[prow], :] * M[prow+1:Nrow, [c]]
                break # next round for pivot
    end end end
    pivot = pivot[2:end] # remove extra initial zero
    vals = [f1 for _ in 1:Ncol] # initialize with ones;
    for prow in length(pivot):-1:1 # going upwards from the last pivoted row
        pc = pivot[prow] # pivoted column
        # overwrite value in pivoted column
```

```
            vals[pc] = -sum(M[prow, pc+1:Ncol] .* vals[pc+1:Ncol])
        end
        @assert all(M * vals .== 0) # Assert that every equation is zero
        return vals
    end
```

C.3. Polinom faktorizálása

Ezt a műveletet a Nemo programkönyvtár segítségével végzem el, ami a Flint programkönyvtárat köti össze a Julia programozási nyelvvel. A Flint könyvtár a faktorizálást a Berlekamp [2], Cantor-Zassenhaus[4], Kaltofen-Shoup faktorizáló algoritmusok egyikével végzi el, a bemenettől függően.

```
#...
Qfactor = factor(Q)
#...
```

C.4. Polinom kiválasztása

Az algoritmus utolsó lépésében kiválasztja a faktorizálás tényezőiből a megfelelőeket. Három fő feltételnek kell megfelelnie egy tényezőnek a kiválasztáshoz. Először is tartalmaznia kell egy x^0y^1 fokú monomot. Másodszor ezen kívül csak x^ky^0 alakú monomok összege lehet. Harmadszor ezekben a tagokban $k \leq d$ egyenlőtlenségnek teljesülnie kell. Ez alapján a kiválasztóprogram:

```
#...
possible_messages = []
for (poly, count) in Qfactor # poly is in the form [α(y - f(x))]
    cy = coeff(poly, [0, 1])
    if cy == 0 continue end # does not have x^0 y^1 monom, skip
    poly = y - poly * inv(cy) # f(x) = y - [α(y - f(x))]/α
    hasy = false # stays false, if poly2 does not have monom with y
    hasd = false # stays false, if the degree poly2 is lower than d
    for (xe, ye) in exponent_vectors(poly)
```

```
            if ye != 0 # has some x^k y^j monom, where j != 0; skip
                hasy = true
                break
            end
            if xe > d # has some x^k y^j monom, where k > d; skip
                hasd = true
                break
            end
        end
        if hasy || hasd continue end
        # poly is a possible polynom
        push!(possible_messages, [coeff(poly, [px, 0]) for px in 0:d])
    end
    return possible_messages
end
```

Irodalomjegyzék

[1] Simeon Ball. On sets of vectors of a finite vector space in which every subset of basis size is a basis. *Journal of the European Mathematical Society*, 14(3):733–748, 2012.

[2] E. R. Berlekamp. Factoring polynomials over finite fields. *The Bell System Technical Journal*, 46(8):1853–1859, 1967.

[3] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017.

[4] David G. Cantor and Hans Zassenhaus. A new algorithm for factoring polynomials over finite fields. *Mathematics of Computation*, 36(154):587–592, 1981.

[5] Claus Fieker, William Hart, Tommy Hofmann, and Fredrik Johansson. Nemo/hecke: Computer algebra and number theory packages for the julia programming language. In *Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computation*, ISSAC '17, pages 157–164, New York, NY, USA, 2017. ACM.

[6] V. Guruswami and M. Sudan. Improved decoding of reed-solomon and algebraic-geometry codes. *IEEE Transactions on Information Theory*, 45(6):1757–1767, 1999.

[7] Venkatesan Guruswami and Alexander Vardy. Maximum-likelihood decoding of reed-solomon codes is np-hard, 2004.

[8] Ivanyos Gábor. *Kódelmélet jegyzet.*

[9] W. B. Hart. Fast library for number theory: An introduction. In *Proceedings of the Third International Congress on Mathematical Software*, ICMS'10, pages 88–91, Berlin, Heidelberg, 2010. Springer-Verlag. http://flintlib.org.

[10] Madhu Sudan. Decoding of reed solomon codes beyond the error-correction bound. *Journal of Complexity*, 13(1):180–193, 1997.

[11] Szőnyi Tamás. *Szimmetrikus struktúrák.*