Sem vložte zadání Vaší práce.

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra počítačových systémů

Diplomová práce

Knihovna funkcí pro program Wolfram Mathematica umožňující snadný návrh bezpečnostních kódů ve výuce

Bc. Jakub Doubek

Vedoucí práce: Ing. Pavel Kubalík, Ph.D.

9. května 2016

Poděkování

Děkuji Ing. Pavlu Kubalíkovi, Ph.D. za odborné konzultace a čas věnovaný mé diplomové práci. Dále děkuji doc. Ing. Aloisu Pluháčkovi, CSc. za laskavé svolení použít v práci jeho obrázky a příklady.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, avšak pouze k nevýdělečným účelům. Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 9. května 2016 .....

České vysoké učení technické v Praze
Fakulta informačních technologií

Odkaz na tuto práci

Doubek, Jakub. *Knihovna funkcí pro program Wolfram Mathematica umožňující snadný návrh bezpečnostních kódů ve výuce.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.

Abstrakt

Práce se zabývá především kódy BCH, RS, LDPC a Hammingovým kódem. V práci je popsán proces kódování a dekódování od tvorby generujících a kontrolních matic a polynomů, přes výpočet syndromů až po opravu chyb. Dále se práce zabývá implementací těchto kódů ve výpočetním prostředí Wolfram Mathematica.

Klíčová slova bezpečnostní kód, samoopravný kód, Hammingův kód, RS kód, BCH kód, LDPC kód, bitflipping algoritmus, Peterson-Gorenstein-Zierler algoritmus, Wolfram

Abstract

This work describes mainly BCH, RS, LDPC and Hamming codes. It describes the process of encoding and decoding from creation of generating and check matrixes and polynomials through syndromes calculation to error correction. Additionally, the implementation of these codes in Wolfram Mathematica is described in this thesis.

Keywords FEC, ECC, RS-code, BCH-code, LDPC code, bitflipping algorithm, Peterson-Gorenstein-Zierler algorithm, Wolfram

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Práce si dává za cíl přiblížit problematiku bezpečnostních kódů a vytvořit prostředí, které by podpořilo a přiblížilo zkoumání bezpečnostních kódů během výuky. Podpořit praktické osahání možností jednotlivých kódů, jejich tvorbu a zkoumání jejich vlastností na základě volby parametrů.

Kapitola 1

Teorie bezpečnostních kódů

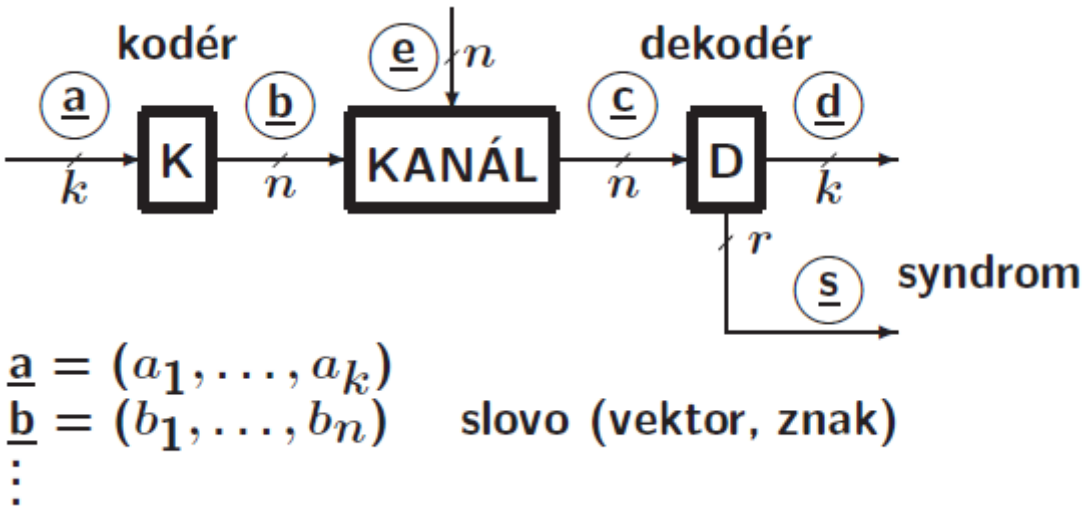
1.1 Chyby a jejich typy

Kanálem označíme prostředí, do kterého nějaká data vložíme a z kterého zase data přijmeme. Avšak ne vždy se podaří, aby vstupní data byla stejná jako výstupní. Může dojít k tomu, že se část dat ztratí - některé znaky z poslané sekvence zmizí. Také se může stát, že v sekvenci znaků se objeví některé znaky navíc, které tam původně nebyly. Dalším typem chyb je situace, kdy dojde k záměně nebo špatnému přečtení znaku. Důležité je, že délka zprávy zůstává stejná, pouze některé znaky nejsou správně rozpoznány. Tato práce se věnuje právě tomuto poslednímu typu chyb. Každý jeden takový nečitelný/zaměněný znak se počítá jako jedna chyba. A protože slova budou pevné délky, lze chyby vyjádřit jako vektor stejné délky, který se přičte k vstupním datům a tím vzniknou data výstupní. A pokud pracujeme s binárními vektory, tak přičtení vektoru odpovídá operaci xor a odpovídá to i operaci odečtení. Zmiňuji to proto, že v práci se bude mluvit o tom, že k chybnému slovu přičteme vektor chyb a tím získáme slovo bez chyb. Jako chyba bude označen každý špatně interpretovaný znak, tedy každý nenulový symbol ve vektoru chyb.

1.2 Bezpečnostní kód

Bezpečnostní kód je souhrnným názvem pro samoopravný a detekční kód. Detekční kód lze použít jako nástroj, který nás upozorní na změnu přenášené informace, pokud je změna dostatečně malá. Samoopravný kód se případnou změnu pokusí opravit, avšak změny, které chceme umět opravovat, nemohou být tak velké jako ty, které nám stačí detekovat. Zpravidla lze opravit ostře menší polovinu chyb, než kolik jich lze detekovat.

Příkladem kódu může být Ascii kódování, ale třeba i Morseova abeceda. Bezpečnostní kód pak musí mít část informace redundantní. V podstatě vždy se jedná o nějaký kontrolní součet.



Obrázek 1.1: Proces kódování a dekódování. Obrázek přebrán z [1]

Na obrázku vidíme proces kódování a dekódování.
Vektor a je původní informace.
Vektor b je zakódovaná informace před přenosem tj. odeslané slovo.
Vektor c představuje přijaté slovo.
Vektor d je informace odpovídající přijatému slovu c.
Písmeno e představuje vektor chyb, ke kterým došlo během přenosu kanálem.
Písmenem s je označen syndrom chyby.
Kanál je cesta (paměť,vysílání), kde může dojít ke změně mezi vstupem a výstupem (změně informace, tj. dostaneme jiné slovo, než které jsme do kanálu poslali - vlivem prostředí kanálu (elektromagnetické záření, fyzické poškození - škrábanec na CD, špatná interpretace scanování dokumentu a pod.))

Kodér přijímá slova délky k a kóduje je na slova délky n. Dekodér přijímá slova délky n, spočítá syndrom, což umožní detekci chyb. V závislosti na nastavení buď pouze oznamuje chyby, nebo je i opravuje. Výstupem je pak slovo délky k.

1.3 Dělení bezpečnostních kódů

Lineární kód délky n a stupně k je lineární podprostor o rozměru k vektorového prostoru GF_q^n, kde GF_q je konečné těleso s q prvky. Jinak řečeno lineární kód je kód, kde lineární kombinace dvou nebo i více kódových slov je opět kódové slovo.

Kódy, kde q=2 se nazývají dvojkové nebo grupové kódy a na ty se tato práce zaměřuje.

Blokový kód zpracovává informační řetězec o délce k a převádí jej na kódové slovo o délce n, přičemž obě tyto délky jsou pro daný kód konstantní. Oproti tomu konvoluční kódy nejsou omezeny délkou přenášené zprávy.

Bezpečnostní kódy lze dělit z více pohledů. Respektive dělení podle „síly“ kódu (počet opravitelných/detekovatelných chyb/shluků) je spíše důsledek nějakého nastavení parametrů nežli nějaký typ kódu. Jedná se tedy o vlastnost kódu. Podobně pokud kód bude postaven na jiné než dvojkové soustavě, týká se dělení spíše tělesa, nad kterým byl kód použit. Někdy může být hranice velmi úzká, mezi tím, co je vlastností kódu a co je důsledkem toho, jak byl kód použit. Výpočetní prostředí často předpokládá použití dvojkové soustavy, a tak některé výpočty nejsou podporovány. Podobně pak prokládané kódy jsou dle mého názoru spíše nějakou metodou aplikovatelnou na širší škálu kódů nežli nějaký konkrétní typ kódu, který se vymezuje strukturou od ostatních. Také dělení kódu na detekční a samoopravný je spíše způsobem použití. Stejně tak nejjednodušší příklady kódů spadají do více kategorií. Nicméně v této práci rozdělím do samostatných částí tyto kategorie kódů: parita, soustava parit, příčná a podélná parita, Hammingovy kódy, rozšířené Hammingovy kódy, kódy generované polynomem, cyklické kódy, BCH, RS, LDPC, Fireův kód.

1.3.1 Parita

Parita je nejjednodušší varianta zabezpečení dat. Rozlišujeme sudou a lichou paritu. Sudá parita znamená, že každé kódové slovo obsahuje sudý počet jedniček. Lichá parita znamená, že každé kódové slovo obsahuje lichý počet jedniček. Tedy za n-bitový vektor dat přidáme ještě jeden bit. Tento bit bude obsahovat znak 1 nebo 0 tak, aby celkový počet jedniček splňoval pravidlo parity. Tento kód nemá žádné opravné schopnosti. Detekovat dokáže pouze jednu chybu. Je pravdou, že tento kód detekuje vlastně libobolný lichý počet chyb, ale pro účely měření síly detekce chyb říkáme, že dokáže detekovat jednu chybu.

Poznámka: Lichá parita nepatří do lineárních kódů. Vezmeme-li totiž dvě slova s lichou Hammingovou váhou a sečteme je, získáme slovo se sudou Hammingovou váhou. Sudá parita lineární je.

1.3.2 Soustava parit

Od detekce jedné chyby pozvolna přejdeme k opravě jedné chyby. Vyjděme z toho, co dokáže parita - to je detekovat jednu chybu ve vektoru pevné délky. Pokud takový vektor rozdělíme na dvě části a pro každou použijeme paritu zvlášť, budeme mít možnost určit, zda je chyba v první či druhé části. Pokud bychom toto vzali do extrému, můžeme vektor rozdělit na jednotlivé znaky a pro každý mít paritu zvlášť. První možností, jak to udělat, je, že zapíšeme odesílané slovo a za něj jednotlivé parity. V takovém případě když zakódu-

Tabulka 1.1: Tabulka interpretace přijaté sekvence znaků v koktavém kódu

přijatá sekvence	interpretace
000	0
001	0
010	0
100	0
011	1
101	1
110	1
111	1

jeme slovo a, tak dostaneme kódové slovo $b = aa$. Takovému kódování říkáme opakovací kód.

Jinou možností je zapsat paritu ihned za každým jednotlivým znakem. Potom vektor (u_1, u_2, u_3) se zakóduje jako $(u_1, u_1, u_2, u_2, u_3, u_3)$. Této variantě říkáme koktavý kód.

Kódové slovo je dvakrát delší než data, ale víme, že kde se shodují paritní bity, tam nenastala chyba. Chyba může být pouze tam, kde se parita neshoduje. Bohužel nevíme, zda je chybný datový nebo příslušný paritní bit. Nemůžeme stále tedy nic opravit. Abychom mohli opravit jednu chybu, potřebujeme data zopakovat ne dvakrát, ale třikrát (ke každému znaku vytvořit dvě parity). V takovém případě se 0 zakóduje na 000 a 1 se zakóduje na 111. Interpretace takové trojice znaků se provede podle tabulky 1.1.

Jinými slovy jde o majoritu. Jako přijatý znak se vezme ten, který ve slově převládá. Tedy opakovací kód, který opakuje třikrát, dokáže opravit jednu chybu. Později se ukáže, že toto řešení je sice funkční, ale velmi neúsporné.

Obecně je soustava parit metoda, při níž spočítáme parity pro určité vybrané podmnožiny znaků z vektoru dat. V podstatě všechny následující kódy jsou nějakou soustavou parit.

1.3.3 Příčná a podélná parita

Je to kód určený pro opravu jedné chyby. Využívá se zde myšlenky uspořádat data do obdélníkového tvaru a spočítat paritu každého sloupce a každého řádku. Pokud detekujeme, ve kterém řádku a kterém sloupci je chyba, můžeme ji jednoduše opravit. Redundance je nižší než u opakovacího a koktavého kódu. Ale stále existují ještě efektivnější kódy. Hammingovy kódy.

1.3.4 Hammingovy kódy

Hammingovy kódy jsou nejúspornější metodou pro opravu jedné chyby ve smyslu počtu přidaných kontrolních znaků. Jedná se o perfektní kód SEC.

Tabulka 1.2: Tabulka parametrů Hammingových kódů

k	1	4	11	26	...	$2^r - 1 - r$
n	3	7	15	31	...	$2^r - 1$
r	2	3	4	5	...	r

Tabulka 1.3: Tabulka parametrů rozšířených Hammingových kódů

k	1	4	11	26	...	$2^{r-1} - r$
n	4	8	16	32	...	2^{r-1}
r	3	4	5	6	...	r

Kódová vzdálenost je 3. V nezkrácené podobě (a tedy s maximální efektivitou) mají Hammingovy kódy parametry uvedené v tabulce 1.2:

1.3.5 Rozšířené Hammingovy kódy

Jedná se o jednoduchou úpravu Hammingova kódu, která o jedna prodlouží kódová slova. Tento kód je pak schopný detekovat o chybu více než originální Hammingův kód. Korekční schopnost rozšířeného kódu zůstává stejná (opraví jednu chybu), avšak dojde-li ke dvěma chybám, je schopný toto alespoň detekovat. Protože všechny syndromy jsou optimálně využity pro korekci i detekci, říkáme, že rozšířený Hammingův kód je kvaziperfektní kód SEC-DED. Parametry rozšířeného Hammingova kódu shrnuje tabulka 1.3.

Obecně lze rozšířit jakýkoli kód s lichou kódovou vzdáleností. Vzhledem k tomu, že se zvýší kódová vzdálenost na sudé číslo, korekční schopnost se nezmění, pouze se zlepší možnosti detekce.

1.3.6 Kódy generované polynomem

Některé lineární kódy lze generovat pomocí polynomu. Výhodou je jistá pravidelnost a s tím spojená nižší složitost kódování a dekódování. Každý takový kód lze vyjádřit ve tvaru generující a kontrolní matice, ale v praxi se pro výpočet maticové násobení vůbec neprovádí. Kóduje a dekóduje se pomocí násobení polynomů a spočítání zbytku po dělení polynomu polynomem. Zpracovává se sériově a většinou se používá pouze k detekci chyb. Dojde-li k chybě, přenos se opakuje.

1.3.7 Cyklické kódy

Známé také jako CRC (z angličtiny Cyclic Redundancy Check). Jsou podmnožinou lineárních kódů generovaných polynomem. Používají se pro zabezpečení proti shluku chyb. Lineární blokový kód nazveme cyklický, pokud všechna slova kódu mají v kódu také slovo, které vznikne cyklickým posunem o jedna vlevo.

1.3.8 BCH kódy

BCH kódy jsou podmnožinou lineárních cyklických kódů. Klíčovou vlastností BCH kódů je možnost v průběhu návrhu kódu přesně kontrolovat počet opravitelných chyb ve výsledném kódu. Další výhodou BCH kódů je jednoduchost jejich dekódování pomocí algebraických metod, známých jako syndrome decoding. BCH kódy jsou používány například v satelitní komunikaci, CD a DVD přehrávačích, pevných discích, flash discích a QR kódech.

1.3.9 RS kódy

Na RS kódy lze pohlížet jako na podmnožinu BCH kódů. RS kód je nebinární cyklický samoopravný kód. V této práci to bude jediný příklad nebinárního kódu. Jeho výhodou je veliká korekční schopnost. Používá se v CD, DVD, Blu-ray discích, pro zabezpečení přenosu pomocí DSL i pro přenos televizního signálu ve formátech DVB a ATSC [2].

1.3.10 LDPC kódy

Výhoda LDPC kódů spočívá v možnosti přenosu dat rychlostí velmi blízkou kapacitě kanálu. Další výhodou je jednoduché dekódování. Velkým plusem je možnost paralelizace. Nevýhodou je časová náročnost kódování. LDPC bývají také kombinovány s jinými kódy, kde jiný kód nahrubo opraví většinu chyb a LDPC poté dohledá zbylé chyby, kterých už není mnoho.

1.3.11 Fireův kód

Je to kód generovaný mnohočlenem. Jeho výhodou je možnost při návrhu jednoduše nastavit velikost shluku chyb, který má kód být schopný detekovat.

1.3.12 Prokládané kódy

Prokládané kódy jsou vhodné jako ochrana proti shluku chyb. Metodou proložení se shluk chyb rozptýlí a je možné ho opravit. Nevýhodou je, že data nemůžeme odesílat, dokud neznáme celý blok dat. Podobná nevýhoda je i po přijetí dat, kdy čtení dat je možné až v okamžiku, kdy máme přijatý celý blok.

Problematiku proložených kódů se pokusím přiblížit na špatně zkopírované stránce textu. Předpokládejme, že kopírovací zařízení udělá na kopii pruh, který není čitelný. Bude-li tento pruh rovnoběžný s textem na papíru, přijdeme o celý řádek textu a bude těžké odhadnout, co na tomto místě původně bylo. Pokud však před kopírováním otočíme stránku o 90 stupňů, budeme mít na kopii pruh kolmo na text. Chyba se projeví na každém řádku textu, ale bude chybět jen malý kousek - jedno písmeno. Budeme-li umět zabezpečit řádek proti jedné chybě, tak potom pomocí proložení jsme schopni chránit text proti shluku chyb o délce m, která se rovná počtu řádků textu. Opravit jednu

Tabulka 1.4: Klony kódu

a		K1			K2			K3		
0	0	0	0	0	0	1	1	1	1	0
0	1	0	1	1	0	0	0	1	0	1
1	0	1	0	1	1	1	0	0	1	1
1	1	1	1	0	1	0	1	0	0	0

chybu na řádku umožní například Hammingův kód, kde každý řádek bude představovat jeden blok kódu.

Na opakovací kód lze pohlížet jako na proložený koktavý kód. Je zřejmé, že má-li při přenosu dojít ke shluku chyb do délky k, jsme z opakovacího kódu schopni zrekonstruovat celou zprávu. U koktavého kódu toto nelze.

1.4 Základní vlastnosti a pojmy

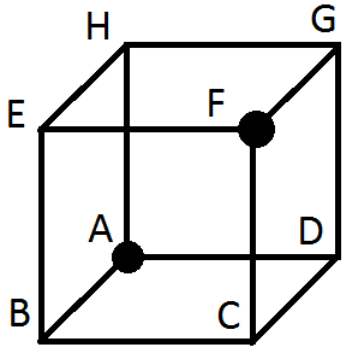
Má-li kód K1 stejné množiny kódových slov jako kód K2, budeme říkat, že K2 je klonem kódu K1. Příklad: kódy K2 i K3 v tabulce 1.4 jsou klonem kódu K1.

Zaměníme-li pořadí znaků v kódu K1 podle permutace π, vzniklý kód K2 budeme nazývat ekvivalentní s kódem K1. Jeho kontrolní a generující matice vzniknou stejnou permutací π z matic kódu K1.

Je-li G generující maticí kódu K1 a zároveň je G kontrolní maticí kódu K2, pak K1 a K2 jsou vzájemně duálními kódy. Kódová slova kódu tvoří nulový prostor duálního kódu.

Jak je psáno v [2], perfektní kód je takový, který má rozdělený celý lineární prostor do (algebraických) koulí s pevným poloměrem a středem v kódových slovech. Tyto koule vyplňují celý prostor. Potom libovolné nekódové slovo patří právě do jedné takovéto koule. U binárních kódů si to lze představit na hyperkrychli. Vzdálenost pak odpovídá počtu hran. A každé nekódové slovo má právě jedno kódové slovo, ke kterému je nejblíže. V tom spočívá princip korekce. Přijmeme-li nekódové slovo, interpretujeme jej jako nejbližší kódové slovo. Perfektní kód SEC je takový kód, který všem nenulovým syndromům přiřazuje význam v podobě právě jedné chyby. Koule pak mají poloměr 1. Grafické zobrazení pro jednoduchý kód ukazuje obrázek 1.2. A a F jsou kódová slova. B,D a H jsou nekódová slova, která spadají do sféry o poloměru 1 kódového slova A. C,E a G jsou nekódová slova, která spadají do sféry o poloměru 1 kódového slova F. A protože těmito sférami je rozdělen celý prostor, kód je perfektní.

U perfektních kódů mají kódová slova mezi sebou Hammingovu vzdálenost rovnu minimální kódové vzdálenosti kódu. Perfektní kód se také někdy nazývá kód těsný. Hammingův kód je perfektní SEC kód.



Obrázek 1.2: Zobrazení lineárního prostoru na hyperkrychli

Co se týče kvaziperfektního kódu, kódová slova mezi sebou mají Hammingovu vzdálenost rovnu minimální kódové vzdálenosti. Ale protože mají Hammingovu vzdálenost sudou, existují zde i nekódová slova, která jsou přesně v polovině vzdálenosti mezi kódovými slovy. Takováto slova nelze opravovat, lze je však použít k detekování chyb. Kvaziperfektní kód SEC-DED je například rozšířený Hammingův kód.

Definice: Necht' A je množina znaků (abeceda). Slovo je konečná posloupnost znaků z množiny A. Počet znaků ve slově je délka slova. Kód K je množina všech slov, která generuje kodér. Prvek kódu K se nazývá kódové slovo. Blokový kód K obsahuje jen slova stejné délky. Binární kód je kód se slovy nad abecedou A = {0, 1}.[3]

Generující matice lineárního kódu K je matice, která v řádcích obsahuje bázi kódu. Vynásobíme-li vektor s odesílanou informací touto maticí, získáme příslušné kódové slovo.

Kontrolní matice je tvořena bází nulového prostoru ke generující matici.

Systematický kód je takový kód, jehož kódová slova obsahují nezměněnou původní informaci doplněnou o paritní bity. Oproti tomu nesystematický kód informační znaky pozmění.

Generující matice systematického kódu má pak tvar: G = $[I_k|F]$ kde I_k je jednotková matice o rozměrech k x k a F je libovolná matice k x r.

Kontrolní matice pak bude vypadat H = $[-F^T|I_r]$, kde I_r je jednotková matice o rozměrech r x r.

1.5 Detekční a korekční schopnost bezpečnostních kódů

Hammingova váha vektoru v je počet jeho nenulových znaků. Hammingova vzdálenost dvou slov v a w je počet pozic, kde jsou znaky odlišné (pro bi-

nární kód je to váha slova v xor w). Kódová vzdálenost (označována kvzd) je minimální Hammingova vzdálenost mezi všemi kódovými slovy. U lineárních kódů určuje opravnou/detekční sílu kódu. Kód opravující 1 chybu se označí SEC (single error correcting). Kód detekující jednu chybu SED (single error detecting). Pro dvě chyby je to DED/DEC (double), pro tři TED/TEC (triple).

Parametr k určuje délku dat před zakódováním, n je délka dat po zakódování. Parametr $r = n - k$ je redundance (nadbytečnost) - počet znaků, o který se zvýší délka dat po zakódování. Jsou to znaky, které nenesou žádná nová data a tím snižují rychlost průchodu dat kanálem.

Proto zavedeme veličinu relativní míra informace, která se spočte jako k/n. Relativní míra informace také bývá označována jako hustota kódu. Podobně pak relativní redundance se spočítá jako r/n.

Syndrom - délka syndromu je r znaků. Ukazuje charakter chyb. Nulový syndrom značí přenos bez chyb, nenulový pak říká, k čemu došlo. Slabší zabezpečení pouze odhalí, že k nějakým chybám došlo, silnější dokáží i přesně určit konkrétní chyby. Syndrom s se spočítá jako $c \cdot H^T$. Ale vzhledem k tomu, že $c = b + e$ a dále $b \cdot H^T = 0$ tak platí, že $s = e \cdot H^T$

Kapitola 2

Rešerše

Hledal jsem výpočetní prostředí podporující práci s maticemi, mnohočleny, Galoisovými tělesy, počítání zbytku po dělení (nad celými čísly i nad polynomy).

Protože práce s bezpečnostními kódy obnáší hodně maticových výpočtů, začal jsem s rešerší v dokumentaci programu MATLAB na stránkách www.mathworks.com. Prostředí podporuje kódování a dekódování Hammingovými kódy, BCH, RS, LDPC. Nenalezl jsem podporu pro Fireův kód. Ale prostředí podporuje práci s mnohočleny i s Galoisovými tělesy. Přestože škola má licence k užívání tohoto programu, více se studenti setkají s programem Wolfram Mathematica, k jehož užívání má škola licence také.

Jako bezplatnou alternativu k MATLABu jsem nalezl octave-online.net. Autoři píší, že je velmi podobný MATLABu a velká část příkazů z MATLABu by měla být podporována i v Octave. Bohužel některé funkce fungující v MATLABu jsou zde ve fázi vývoje a nejsou zatím podporovány. Také stránky s manuálem nebyly dostupné, takže jsem zkoušel příkazy z MATLABu, některé byly podporovány, jiné alespoň vypisovaly, že zatím nejsou podporovány.

Dále jsem se setkal s několika aplety napsanými v Javě - nalezl jsem spíše simulátory jednotlivých kódů s velmi omezenými možnostmi. Někdy ani nebylo možné vložit vlastní generující polynom, program pouze nabídl vybrat z několika předdefinovaných. Tyto aplety jsou často zastaralé, nepodporované kvůli bezpečnostnímu riziku a v současné době často zakazované webovými prohlížeči.

Co mě zaujalo, byla zmínka o WebMathematice. WebMathematica umožňuje přidat na webové stránky interaktivní prostředí pro výpočty a vizualizace. Z popisu odhaduji, že se jedná o něco podobného jako je webová stránka wolframalpha.com, ale s možností předdefinovat výpočty, nastavovat parametry pomocí příslušného táhla, ukládat a sdílet výsledky a podobně. Více se lze dozvědět na webových stránkách
http://www.wolfram.com/products/webmathematica/whatis.html.

Další velmi silné výpočetní prostředí je Sage, které je navíc bezplatně k dis-

Tabulka 2.1: Tabulka podporovaných operací

	MATLAB	Octave-online	Sage
(De)kódování lineárního kódu definovaným G/H	ANO	ANO	ANO
(De)kódování cyklického kódu zadaným G(x)	ANO	ANO	ANO
Tvorba G,H na základě G(x) a n	ANO	ANO	ANO
Tvorba G(x) pro cyklický kód na základě n,k	ANO	ANO	ANO
Všechny G(x) pro cyklický kód na základě n,k	ANO	ANO	NE
Podpora tvorby GF	ANO	ANO	ANO
Podpora převodu polynomů na formát vektoru	ANO	NE	NE
Hammingův kód			
Tvorba G,H na základě r	ANO	ANO	ANO
Tvorba tabulky syndromů	ANO	ANO	ANO
Výpočet syndromu	ANO	ANO	ANO
Počítání kódové vzdálenosti z G(x)/G/H.	ANO	ANO	ANO
BCH			
G(x) pro BCH kód na základě vloženého n a k	ANO	ANO	ANO
Počítání kódové vzdálenosti pro BCH kód.	ANO	NE	ANO
Počítání polynom modulo polynom	ANO	ANO	ANO
RS			
Kódování a dekódování	ANO	ANO	ANO
Tvorba G(x) pro RS kód	ANO	ANO	ANO
Zkracování kódů RS	ANO	NE	ANO
LDPC			
Podpora tvorby LDPC dle standardu DVB-S.2	ANO	NE	NE

pozici. Nejenže podporuje spoustu operací pro práci s bezpečnostními kódy, ale mnoho kódů je již implementováno. Pro tvorbu kódu většinou používá jeden konkrétní generátor, ale alespoň podporuje hledání automorfismů a permutací. Interpreter Sage je k dispozici online na adrese https://cloud.sagemath.com, je však nutná registrace.

Kapitola 3

Analýza

3.1 Parita

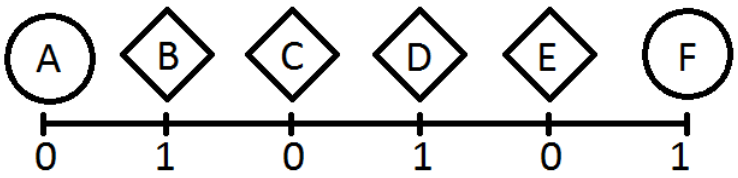
Tvrzení: podle parity lze poznat, zda při přenosu došlo k sudému počtu chyb (parita souhlasí) či lichému počtu chyb (parita nesouhlasí). Důkaz:

- Paritu lze spočítat jako Hammingovu váhu modulo 2 - záleží pouze na tom, zda je Hammingova váha lichá nebo sudá.
- Každá chyba v binárním kódovém slovu změní Hammingovu váhu vektoru přesně o jedna. Buď dojde k změně z 0 na 1 a Hammingova váha se o jedna zvýší, nebo se 1 změní na 0 a Hammingova váha se o jedna zmenší. Všimněme si, že toto „počítání chyb“ bude fungovat i pro případ, že dojde k chybě v paritním bitu.

Proto lze paritu použít na detekci jedné chyby. Dvě chyby už detekovat nelze, protože parita bude stejná jako v případě nula chyb.

Další využití parity je pro rozšířené kódy. Kódům s lichou kódovou vzdáleností lze kódovou vzdálenost o jedna zvýšit právě pomocí parity. Mají-li dvě kódová slova lichou Hammingovu vzdálenost, musí jedno z nich mít lichou a druhé sudou Hammingovu váhu. Tím se k jednomu přidá jako parita 0 a k druhému 1. A právě o tento paritní bit se jejich vzdálenost o jedna zvýší.

Na rozšíření kódu můžeme také pohlížet jako na další kódování. Ke kódovým slovům rozšiřovaného kódu K přidáme paritu. Parita, jak je popsáno výše, dokáže určit, zda došlo k lichému či sudému počtu chyb. Nyní ukáži, jak to pomůže zvýšit o jedna počet chyb, které jsme schopni detekovat. Tím bude dokázáno i to, že se o 1 zvedla kvzd. Vycházíme z liché kódové vzdálenosti tedy kvzd $= 2x + 1$, kde x je celé číslo. Případ kdy $x = 2$ bude odpovídat obrázku 3.1. Slova A a F jsou kódová, slova B,C,D,E nekódová. Pod slovy je ještě uveden součet očekávané a skutečné parity za předpokladu, že původní přenášené slovo je A. Pokud je tento součet 1, parita neodpovídá a značí to chybu. Slova B,C,D,E jsme byli schopni detekovat i bez rozšíření, protože nejsou kódová.



Obrázek 3.1: Cesta liché délky mezi dvěma kódovými slovy

Dojde-li k $2x+1$ chybám, vznikne však kódové slovo F, a proto běžně nejsme schopni tento počet chyb odhalit. Jenže v případě rozšířeného kódu, dojde-li k $2x+1$ chybám, z parity zjistíme, že došlo k lichému počtu chyb, a tedy přenos nemůže být bez chyby. Takto tedy detekujeme i $2x+1$ chyb.

Počítat paritu nebude problém, stačí sečíst modulo 2 všechny znaky. Výsledek takového součtu bude právě paritní bit, o který rozšíříme vstupní data. Generující matice bude jednotková matice rozšířená zprava o sloupec jedniček.

$$\left(\begin{array}{cccc|c} 1 & 0 & \cdots & 0 & 1 \\ 0 & 1 & \cdots & 0 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 1 \end{array}\right)$$

Kontrola, že nedošlo k chybě, se provede tak, že se sečtou znaky přijatého slova. V případě, že součet bude 0, tak při přenosu nedošlo k pozměnění kódového slova. V případě, že součet je 1, kódové slovo bylo pozměněno. Kontrolní matice tedy budou samé jedničky (bude jich n).

$$\left(\begin{array}{cccc} 1 & 1 & \ldots & 1 \end{array}\right)$$

3.2 Příčná a podélná parita

Tento kód je navržen pro opravu jedné chyby. Vepíšeme-li slovo do obdélníkového tvaru a po řádcích spočteme podélnou paritu a po sloupcích příčnou paritu, umožní nám to zjistit „souřadnice“, kde se vyskytuje chyba. Spočtení parit si ukážeme na příkladu, který je uveden v [1].

$k = 4$ rozložíme na $2 \cdot 2$

$a = (a_1, a_2, a_3, a_4)$

a_1	a_2	$p_1 = a_1 \oplus a_2$
a_3	a_4	$p_2 = a_3 \oplus a_4$
$p_3 = a_1 \oplus a_3$	$p_4 = a_2 \oplus a_4$	

$n = k + \#\text{řádků} + \#\textit{sloupců} = 4 + 2 + 2 = 8$

$b = (a_1, a_2, a_3, a_4, p_1, p_2, p_3, p_4)$

kvzd $= 3$

Generující matice kódu Příčná a podélná parita bude vypadat takto

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

Kontrolní matice kódu Příčná a podélná parita má tvar

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

Dále je ještě možné provést rozšíření. Pokud spočteme paritu parit (nezáleží na tom zda příčných či podélných-bude stejná $p_5 = p_1 + p_2 = p_3 + p_4 = a_1 + a_2 + a_3 + a_4$), zvýší se kódová vzdálenost na 4 a to nám umožní detekovat 2 chyby. S rozšířením tedy bude kód vypadat takto [1]:

$k = 4$

a_1	a_2	$p_1 = a_1 \oplus a_2$
a_3	a_4	$p_2 = a_3 \oplus a_4$
$p_3 = a_1 \oplus a_3$	$p_4 = a_2 \oplus a_4$	$p_5 = p_1 \oplus p_2$

$b = (a_1, a_2, a_3, a_4, p_1, p_2, p_3, p_4, p_5)$
$n = 9$
kvzd $= 4$

Tvar generující matice rozšířeného kódu Příčná a podélná parita.

$$\left(\begin{array}{cccccccc|c} 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \end{array}\right)$$

Kontrolní matice rozšířeného kódu Příčná a podélná parita má tvar

$$\left(\begin{array}{cccccccc|c} 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ \hline 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{array}\right)$$

K implementaci nepotřebujeme žádné další nástroje než ty z kapitoly Parita 3.1.

3.3 Hammingův kód

Kontrolní matice Hammingova kódu (k,n) bude mít r řádků a n sloupců. Ve sloupcích bude mít všechny nenulové r-bitové kombinace. Každý sloupec bude jiný a jediná nevyužitá r-bitová kombinace budou samé nuly. Lze takto rovnou vytvořit systematický kód, když v pravé části matice vytvoříme jednotkovou matici dimenze r. Generující matice se poskládá nejjednodušeji právě pokud budeme mít kód systematický. V takovém případě platí:

$$H = (F|I_r) => G = (I_k| - F^T). \tag{3.1}$$

Budeme-li z jakéhokoli důvodu mít kontrolní matici v jiném tvaru, pokusíme se pomocí Gaussovy eliminační metody s následným zpětným chodem požadovaný tvar zajistit. Podaří-li se to, generující matice vzniklá pomocí vzorečku 3.1 bude fungovat i pro původní tvar kontrolní matice. Protože obě matice popisují tentýž prostor.

Horší situace nastane, když matice H nelze převést do tvaru ve vzorečku 3.1. Pak je možné, jak je uvedeno v [1], si sloupce matice přeházet, avšak je třeba si zapamatovat permutaci π, jak byly jednotlivé sloupce zpřeházeny. Takovou matici označme $H^{\mathbb{D}}$. Pro $H^{\mathbb{D}}$ najdeme $G^{\mathbb{D}}$ podle vzorce 3.1. Matici G pak dostaneme, když na $G^{\mathbb{D}}$ použijeme opačnou permutaci π^{-1}. Jinými slovy: pro každý lineární kód K1 existuje systematický lineární kód K2, který má pouze jiné pořadí znaků v kódovém slově. Pokud zpřeházíme znaky tak, že z K1 dostaneme K2 a vytvoříme generující matici pro K2, je potřeba znaky zase zpřeházet zpět, abychom dostali generující matici pro K1.

V příkladu níže, který je také převzat z [1] jsme nejprve přesunuli pátý sloupec na druhé místo 3.2. Po spočtení kontrolní matice je potřeba vrátit druhý sloupec na páté místo 3.3. Odpovídající permutace jsou vyjádřeny v 3.4 a 3.5.

$$G = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \end{pmatrix} \rightarrow \pi \rightarrow G^{\mathbb{D}} = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 \end{pmatrix} \tag{3.2}$$

$$H = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix} \leftarrow \pi^{-1} \leftarrow H^{\mathbb{D}} = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \end{pmatrix} \tag{3.3}$$

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 3 & 4 & 5 & 2 \end{pmatrix} \tag{3.4}$$

$$\pi^{-1} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 3 & 4 & 5 & 2 \end{pmatrix} \tag{3.5}$$

Metodu s přehazováním sloupců lze použít současně s Gaussovou eliminační metodou. Tyto metody pro hledání generující matice z kontrolní jsou obecné a platí i pro ostatní maticemi tvořené kódy.

Nové požadavky pro implementaci: Hammingův kód v systematickém tvaru potřebuje počítat transponovanou matici. Pro nesystematickou verzi bude potřeba navíc Gaussova eliminační metoda, prohazování řádků a sloupců matice. Přičítání řádků k jinému řádku. Rozšíření matice o vektor a spojení více matic v jednu.

3.4 Rozšířený Hammingův kód

Rozšířením Hammingova kódu (n,k) vznikne rozšířený Hammingův kód (n+1,k). Důsledkem rozšíření bude o jedna delší kódové slovo, ale zvýší se o jedna i kódová vzdálenost. Ze tří na čtyři. Což umožní buď detekovat až tři chyby(TED) nebo jednu chybu opravovat a dvě detekovat (SEC-DED).
Jak bude vypadat kontrolní matice rozšířeného Hammingova kódu, ukazuje obrázek níže:

$$\left(\begin{array}{ccc|c} & & & 0 \\ & H^* & & \vdots \\ & & & 0 \\ \hline 1 & \cdots & 1 & 1 \end{array}\right)$$

kde H^* je kontrolní matice rozšiřovaného Hammingova kódu.
Odpovídající generující matice bude mít tvar:

$$\left(\begin{array}{ccc|c} & & & p_1 \\ & G^* & & \vdots \\ & & & p_k \end{array}\right)$$

kde G^* je generující matice rozšiřovaného Hammingova kódu a p_x je sudá parita pro x-tý řádek.

Interpretace syndromu: nejprve zopakujme, že rozšířené Hammingovy kódy jsou SEC-DED. Z kontrolní matice je vidět, že syndrom bude mít navíc jeden bit. Ten ukazuje, zda při přenosu došlo k sudému nebo lichému počtu chyb. Bude-li paritní bit souhlasit, došlo k sudému počtu chyb. Tedy máme-li ostatní bity syndromu nulové, přenos proběhl v pořádku. Nejsou-li ostatní bity syndromu nulové, přijaté slovo je chybné a protože má sudý počet chyb, nemůže být opraveno. Dojde pouze k detekci.

Situace, kdy paritní bit nesouhlasí, také rozdělíme na dva případy. Protože jediný lichý počet chyb, který kód detekuje nebo opravuje, je jedna, předpokládáme, že došlo k jedné chybě. Pokud ostatní bity syndromu jsou nenulové, opravíme příslušnou chybu tak, jak bychom to udělali bez rozšíření. Pokud však jsou ostatní bity syndromu nulové, značí to, že v části přijatého slova příslušící kódu bez rozšíření chyba nenastala. Došlo tedy k jediné chybě a to v bitu rozšiřujícím kód. Situaci shrnuje tabulka 3.1:

Tabulka 3.1: Tabulka interpretace syndromu rozšířeného Hammingova kódu

bity syndromu 1 až r-1	r	interpretace
0...0	0	slovo bez chyb
0...0	1	chyba v rozšiřujícím bitu => není třeba opravovat
nenulový	1	detekce jedné chyby a korekce
nenulový	0	detekce dvou chyb

3.5 Kódy generované polynomem

V kódech generovaných polynomem budeme pracovat s mnohočleny nad konečným tělesem GF(q^m), kde q je prvočíslo. U binárních kódů tedy s tělesem GF(2^m).

Počítání s GF(q^m) vyžaduje definovat primitivní polynom stupně r. Je to polynom, který nám rozdělí polynomy do tříd ekvivalence podobně jako prvočíslo rozdělí celá čísla při počítání v Z_p. Všechny polynomy P(x), které mají stupeň alespoň r, se ireducibilním polynomem vydělí a použije se pouze zbytek po dělení. Takových ireducibilních polynomů stupně r může existovat více. Těleso GF(q^m) % IP_1 je izomorfní s tělesem GF(q^m) % IP_2,kde IP_1 a IP_2 jsou různé ireducibilní polynomy. Nelze proto z hlediska nějakých vlastností upřednostnit jeden ireducibilní polynom před druhým. Jako standard pro sjednocení tvorby GF se používají Conwayovské polynomy. Conwayovský polynom má tu vlastnost, že je minimální z lexikografického hlediska, přičemž platí, že vyšší řády se píší více vlevo. Hledat lze hrubou silou, přičemž u binárního tělesa víme, že Conwayovský polynom musí začínat i končit jedničkou a nemůže mít sudý počet jedniček.

Máme mnohočlen $C(x) = c_{n-1}x^{n-1} + ... + c_1x + c_0$ kde c_i je z GF(2) a x je neznámá. Zápis můžeme zkrátit tak, že se mocniny x vynechají a píšeme pouze koeficienty. Pak je ale potřeba psát mnohočlen systematicky od nejvyšších mocnin k nejnižším a zapisovat i ty mocniny, u kterých je násobek nulový a takový zápis dodržovat. Někdy se totiž zapisují koeficienty mocnin od nejnižších po nejvyšší (Wolfram, případně pro některé funkce je sestupné pořadí koeficientů praktičtější), a proto je vždy nutné vědět, jaké konvence se v tom kterém zápise dodržují.

Jako zkrácený zápis označme uspořádanou n-tici $c_{n-1}, ..., c_0$, ve které ještě navíc odebereme oddělovací čárky. Tyto dva prostory jsou pro účely našich výpočtů (operace sčítání modulo a násobení skalárem) izomorfní.

Reprezentace slov tedy bude vypadat:
$a = (a_{k-1}, ..., a_1, a_0)$ což odpovídá $A(x) = a_{k-1}x^{k-1} + ... + a_1x + a_0$
$b = (b_{n-1}, ..., b_1, b_0)$ což odpovídá $B(x) = b_{n-1}x^{n-1} + ... + b_1x + b_0$

Příklad: máme-li n=7, pak mnohočlenu $x^5 + x^3 + x + 1$ odpovídá zkrácený zápis 0101011.

Generovací mnohočlen bude mít tvar $G(x) = g_{r-1}x^{r-1} + ... + g_1x + g_0$ n=max deg B(x)+1 k=max deg A(x)+1 r=deg G(x) Takže opět platí n=k+r

Kódování probíhá podle vzorce $B(x) = A(x) \cdot G(x)$.

Příklad: G(x)=$x^3 + x + 1$
a = 1010 A(x)=$x^3 + x$
$B(x) = A(x) \cdot G(x) = x^6 + x^3 + x^2 + x$
tedy $b = 1001110$

Můžeme si všimnout, že tento způsob kódování není systematický. Nazýváme ho kód 1. typu [1]. Odpovídající generující matice by vypadala takto:

$$G = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{pmatrix} \tag{3.6}$$

Kód 2. typu je klonem kódu 1. typu, ale je systematický. Tedy vstupní data budou nezměněna na výstupu a za nimi bude dopočten kontrolní součet tak, aby zbytek po dělení byl nula. To můžeme provést tím, že slovo zprava doplníme nulami tak, aby celková délka byla n (odpovídá násobení x^{posun}). Spočteme zbytek po dělení generujícím polynomem taktovéhoto slova a tento zbytek od slova odečteme.

Tomu odpovídá následující generující matice

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{pmatrix} \tag{3.7}$$

Syndrom u kódů generovaných polynomem se spočítá jako $S(x) = C(x)\%G(x) = [B(x)+E(x)]\%G(x) = B(x)\%G(x)+E(x)\%G(x) = 0+E(x)\%G(x) = E(x)\%G(x)$
A tedy syndrom zavisí pouze na chybách.

Rozdělme nyní shluk chyb na tvar chyb a pozici. $E(x) = E'(x) \cdot x^j$, kde x nedělí beze zbytku $E'(x)$, E' je tvar chyb a j je jejich pozice. Délka shluku chyb $L = degE' + 1$.

Pokud $E'(x)$ není 0 a zároveň deg $E'(x) <$ deg $G(x)$, pak $E'(x)$ % G(x) nemůže být 0. Z toho plyne, že lze zjistit libovolný shluk chyb délky $L \leq r = degG(x)$ [1].

Pro implementaci bude potřeba umět počítat zbytek po dělení polynomu polynomem, násobit polynomy a převádět mezi reprezentací polynomů a uspořádaných n-tic(kódových slov).

3.6 Cyklické kódy

Kód generovaný polynomem bude cyklický právě tehdy, když generující polynom dělí beze zbytku polynom $x^n + 1$. Cyklický kód C(n,k) obsahuje právě

jeden generující polynom G(x) stupně n-k. Zpravidla je to nejmenší nenulový polynom, kterým když vydělíme polynom $x^n + 1$, tak bude podíl beze zbytku. Pojdmě se podívat na souvislost mezi generující maticí a generujícím polynomem. Máme generující polynom ve tvaru 3.8.

$$G(x) = g_r(x)^r + g_{r-1}(x)^{r-1} + \cdots + g_0 \tag{3.8}$$

Odpovídající generující matice k polynomu 3.8 vznikne zapsáním koeficientů polynomu do řádky a doplněním nulami. Každý další řádek matice vznikne cyklickým posunem řádku nad ním o jedna vpravo. Tvar takové matice ukazuje bod 3.9.

$$G = \begin{pmatrix} g_r & g_{r-1} & \cdots & g_0 & 0 & 0 & 0 \\ 0 & g_r & g_{r-1} & \cdots & g_0 & 0 & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & g_r & g_{r-1} & \cdots & g_0 \end{pmatrix} \tag{3.9}$$

Podíl $H(x) = (x^n+1) : G(x)$ nazveme kontrolním polynomem. První řádek kontrolní matice 3.11 vznikne zapsáním koeficientů kontrolního polynomu 3.10 a následné doplnění nulami. Na rozdíl od tvorby generující matice je však třeba tyto koeficienty zapisovat od nejnižšího řádu. Další řádky matice se opět vytvoří cyklickým posuvem vpravo.

$$H(x) = h_k(x)^k + h_{k-1}(x)^{k-1} + \cdots + h_0 \tag{3.10}$$

$$H = \begin{pmatrix} h_0 & h_1 & \cdots & h_k & 0 & 0 & 0 \\ 0 & h_0 & h_1 & \cdots & h_k & 0 & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & h_0 & h_1 & \cdots & h_k \end{pmatrix} \tag{3.11}$$

Kódování slova probíhá pomocí vynásobení Generujícím polynomem. Tedy $B(x) = A(x) \cdot G(x)$. Nebo v případě systematického kódu $B(x) = x^r \cdot A(x) + x^r \cdot A(x)\%G(x)$.

Ať už je kód systematický či nikoli, lze v obou případech použít stejný kontrolní polynom pro zjištění syndromu. V maticovém pojetí kódu platí, že syndrom $s = c.H^T = e.H^T$. Zde můžeme syndrom spočítat pomocí vzorce 3.12.

$$S(x) = C(x)\%G(x) \tag{3.12}$$

Ale v praxi se používá jiná možnost. Zavedeme nyní pojem součinový syndrom $S^{\times}(x)$. Ten se vypočítá pomocí kontrolního polynomu dle vzorce 3.13.

$$S^{\times}(x) = C(x) \cdot H(x)\%x^{n-1} \tag{3.13}$$

Součinový syndrom není stejný jako syndrom, ale obsahuje stejnou informaci. Tyto dva syndromy jsou totiž kongruentní [4]. Pokud je syndrom nulový, data jsou nezměněna. Je-li syndrom nenulový, kód detekoval, že došlo ke změně dat. Opravu chyb rozeberu na cyklických kódech BCH v následující kapitole.

Tabulka 3.2: Kódy generované jednotlivými možnými generátory

Označení kódu	polynom	zkrácený zápis	množina kódových slov
K1	1	001	000,001,010,011,100,101,110,111
K2	$x+1$	011	000,011,101,110
K3	x^2+x+1	111	000,111
K4	0	000	000

3.7 Binární kódy BCH

Binární kódy BCH jsou cyklické kódy, které garantují určitou minimální vzdálenost δ mezi kódovými slovy. Jak už jsem psal dříve, u cyklických kódů platí, že mnohočlen x^n-1 musí být beze zbytku dělitelný generátorem kódu. Podíváme-li se na to trochu blíže, můžeme polynom x^n-1 rozložit na součin jednotlivých minimálních polynomů. Minimální polynom je monický polynom, který je ireducibilní. Monický polynom je takový, který má u nejvyššího řádu koeficient 1. Potom každý cyklický kód délky n je generován součinem libovolného počtu těchto minimálních polynomů [9].

Příklad:
Máme těleso GF(2),
n=3.
Polynom $x^3-1=x^3+1$ a jeho rozklad je $(x+1)(x^2+x+1)$.
Monickými děliteli jsou $1,(x+1),(x^2+x+1)$ a součin $(x+1)(x^2+x+1)$, který odpovídá 0 po modulení x^3+1.

Můžeme mít tedy 4 kódy délky 3. Avšak uvidíme, že některé jsou nepoužitelné. Kódy generované jednotlivými polynomy shrnuje tabulka 3.2 Z tabulky je zřejmé, že kód K1 má 8 kódových slov a tedy nemá žádná nekódová. Redundance je nulová a výstup je stejný jako vstup. Což se dalo od násobení jedničkou očekávat. K4 má jediné kódové slovo, a to nulové. Tedy násobení nulou také nemá smysl. Ostatní kódy jsou použitelné. Kód K2 odpovídá zabezpečení sudou paritou. Kód K3 pak opakovacímu a koktavému kódu(3,1).

Podívejme se nyní na kód délky n = 7.
Máme těleso GF(2),
n=7.
Polynom $x^7-1=x^7+1$ a jeho rozklad je $(x+1)(x^3+x+1)(x^3+x^2+1)$.
Monickými děliteli jsou $1,(x+1),(x^3+x+1),(x^3+x^2+1),(x^4+x^3+x^2+1),(x^4+x^2+x+1),(x^6+x^5+x^4+x^3+x^2+x+1)$ a součin $(x+1)(x^3+x+1)(x^3+x^2+1)$, který odpovídá 0 po modulení x^7+1. Máme 8 možných generátorů. O generátorech G=0 a G=1 víme, že nám dají nepoužitelné kódy. Dále máme generátory $(x^4+x^3+x^2+1),(x^4+x^2+x+1),(x^6+x^5+x^4+x^3+x^2+x+1)$. Tyto tři generátory nejsou ireducibilní. Jsou složené a kódům generovaným takto složenými generátory se říká BCH kódy. Zbývají generátory $(x+1),(x^3+x+1)a(x^3+x^2+1)$, tvoří duální kódy k BCH kódům uvedeným výše. Platí tedy, že výsledkem součinu generátoru kódu s generátorem duálního kódu je nula

modulo $x^n + 1$.

Nyní přejdeme k tvorbě BCH kódů se zaručenou kódovou vzdáleností. Aby kód měl zaručenou minimální kódovou vzdálenost δ, musí platit následující věta.

Pro všechny x z množiny $\{\alpha, \alpha^2, ..., \alpha^{\delta-1}\}$ je G(x) = 0, kde α je primitivní prvek tělesa $GF(2^n)$. Generátor pak bude mít tvar:
G(x) = LCM$(M_{\#1}(x), M_{\#2}(x), ..., M_{\#\delta-1}(x))$. LCM je nejmenší společný násobek.
$M_{\#i}(x)$ je minimální polynom prvku α^i. Prvkům $\{\alpha, \alpha^2, ..., \alpha^{\delta-1}\}$ se říká generovací kořeny.

Nyní přejdeme k počítání minimálního polynomu pro prvek $\beta = \alpha^i$.
$M_{\#i}(x) = M_\beta(x) = (x - \beta)(x - \beta^p)...(x - \beta^{(p^{j-1})})$.

Pro kódování a dekódování se sice matice nepoužívají, ale pro lepší porozumění struktuře kódu ukažme, jak by vypadala kontrolní matice, která je ještě pro přehlednost rozdělená do podmatic H_1 až $H_{\delta-1}$:

$$H = \begin{pmatrix} H_1 \\ H_2 \\ \vdots \\ H_{\delta-1} \end{pmatrix} = \begin{pmatrix} \alpha^{n-1} & \alpha^{n-2} & \cdots & \alpha & 1 \\ (\alpha^2)^{n-1} & (\alpha^2)^{n-2} & \cdots & \alpha^2 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ (\alpha^{\delta-1})^{n-1} & (\alpha^{\delta-1})^{n-2} & \cdots & \alpha^{\delta-1} & 1 \end{pmatrix} \tag{3.14}$$

Z matice by se ještě odstranily lineárně závislé řádky. Vzhledem k tomu, že $M_{\#i}(x) = M_{\#2i}(x) = M_{\#4i}(x) = ...$, můžeme rovnou při tvorbě z matice 3.14 vynechat všechny sudé řádky a zkonstruovat rovnou tvar

$$H = \begin{pmatrix} H_1 \\ H_3 \\ \vdots \\ H_{\delta-2} \end{pmatrix} = \begin{pmatrix} \alpha^{n-1} & \alpha^{n-2} & \cdots & \alpha & 1 \\ (\alpha^3)^{n-1} & (\alpha^3)^{n-2} & \cdots & \alpha^3 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ (\alpha^{\delta-2})^{n-1} & (\alpha^{\delta-2})^{n-2} & \cdots & \alpha^{\delta-2} & 1 \end{pmatrix} \tag{3.15}$$

Příklad na tělese GF(16) s počítáním modulo $z^4 + z + 1$. Primitivní prvek $\alpha = z$. V tabulce 3.3 máme všechny prvky generované generujícím prvkem. Od i=15 se prvky začínají opakovat.

Množina minimálních mnohočlenů tedy je:

- $M_{\#0}(x) = x + 1$
- $M_{\#1}(x) = x^4 + x + 1$
- $M_{\#3}(x) = x^4 + x^3 + x^2 + x + 1$
- $M_{\#5}(x) = x^2 + x + 1$
- $M_{\#7}(x) = x^4 + x^3 + 1$

Tabulka 3.3: Nenulové prvky tělesa GF(16) a jejich minimální mnohočleny

i	z^i	zkrácený zápis	hexadecimální zápis	minimální mnohočlen
0	1	0001	1	$x+1$
1	z	0010	2	x^4+x+1
2	z^2	0100	4	x^4+x+1
3	z^3	1000	8	x^3+x^2+x+1
4	$z+1$	0011	3	x^4+x+1
5	z^2+z	0110	6	x^2+x+1
6	z^3+z^2	1100	C	x^3+x^2+x+1
7	z^3+z+1	1011	B	x^3+x+1
8	z^2+1	0101	5	x^4+x+1
9	z^2+z+1	0111	A	x^3+x^2+x+1
10	z^3+z^2+z	1110	7	x^2+x+1
11	z^3+z+1	1011	E	x^3+x+1
12	z^3+z^2+z+1	1111	F	x^3+x^2+x+1
13	z^3+z^2+1	1101	D	x^3+x+1
14	z^3+1	1001	9	x^3+x+1
15	1	0001	1	$x+1$

Pro konstrukci kódu s $\delta = 3$ použijeme G(x) $= M_{\#1}(x) = x^4 + x + 1$.

Pro konstrukci kódu s $\delta = 5$ použijeme G(x) $= M_{\#1}(x) \cdot M_{\#3}(x) = (x^4 + x + 1)(x^4 + x^3 + x^2 + x + 1) = x^8 + x^7 + x^6 + x^4 + 1$.

Kód s $\delta = 7$ vznikne při G(x) $= M_{\#1}(x) \cdot M_{\#3}(x) \cdot M_{\#5}(x) = (x^4 + x + 1)(x^4 + x^3 + x^2 + x + 1)(x^2 + x + 1) = x^{10} + x^8 + x^5 + x^4 + x^2 + x + 1$.

Kód s $\delta = 9$ vznikne při G(x) $= LCM(M_{\#1}(x) \cdot M_{\#3}(x) \cdot M_{\#5}(x) \cdot M_{\#7}(x)) = (x^4 + x + 1)(x^4 + x^3 + x^2 + x + 1)(x^2 + x + 1)(x^4 + x^3 + 1) = x^{14} + x^{13} + x^{12} + x^{11} + x^{10} + x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1$.
Poslední zmíněný kód generuje slova (111111111111111) a (000000000000000). Zde je dobře vidět, že kódová vzdálenost je 15, přestože minimální kódová vzdálenost je 9.

Kódování probíhá standartně. Slovo se vynásobí generátorem. Při dekódování se nejprve spočítají syndromy. Ty lze získat jako zbytek po dělení jednotlivými monickými polynomy generátoru.

$$S_i(x) = C(x)\%M_{\#i}(x) \tag{3.16}$$

Pro syndromy také platí

$$S_i(x) = C(\alpha^i) \tag{3.17}$$

Jsou-li syndromy nulové, dekódování tímto končí - přenos proběhl bez chyby. Pokud syndromy nulové nejsou, a my chceme chyby opravovat, mu-

síme zjistit kolik chyb se do přenášeného slova dostalo a kde tyto chyby jsou. Obecně by bylo ještě potřeba vypočítat velikost chyby, ale protože pracujeme s binárním kódem, chyba bude vždy velikosti jedna. Máme vektor chyb ve tvaru 3.18

$$E(x) = x^{i_1} + x^{i_2} + \cdots + x^{i_\nu} \tag{3.18}$$

Kde ν je počet chyb. Nejprve musíme nalézt mnohočlen pozic označovaný také jako lokátor. Ten se značí $\Lambda(x)$ a má tvar 3.19.

$$\Lambda(x) = \lambda_0 + \lambda_1 x + \lambda_2 x^2 + \cdots + \lambda_\nu x^\nu \tag{3.19}$$

Ke spočítání lokátoru lze použít algoritmus Peterson-Gorenstein-Zierler nebo algoritmus Berlekamp-Massey.

V této práci jsem si vybral algoritmus Peterson-Gorenstein-Zierler. Ten řeší problém jako soustavu rovnic a řeší je hrubou silou. Nejprve ze syndromů sestrojíme matici Θ_ν.

$$\Theta_\nu = \begin{pmatrix} s_1 & s_2 & \cdots & s_\nu \\ s_2 & s_3 & \cdots & s_{\nu+1} \\ \vdots & \vdots & \vdots & \vdots \\ s_\nu & s_{\nu+1} & \cdots & s_{2\nu-1} \end{pmatrix} \tag{3.20}$$

Protože zatím neznáme počet chyb, dosadíme za ν maximální opravitelné množství chyb. Poté sepočítáme determinant matice Θ_ν. V případě, že je determinant nulový, skutečných chyb ve zprávě je méně. Proto budeme hodnotu ν snižovat tak dlouho, dokud se nám nepodaří zkonstruovat matici Θ_ν s nenulovým determinantem. Když budeme mít matici Θ_ν s nenulovým determinantem, lze spočítat její inverzi. Inverzní matici totiž použijeme k dopočítání koeficientů lokalizačního polynomu ve vzorci 3.21.

$$\begin{pmatrix} \lambda_\nu \\ \lambda_{\nu-1} \\ \vdots \\ \lambda_1 \end{pmatrix} = {\Theta_\nu}^{-1} \begin{pmatrix} -s_{\nu+1} \\ -s_{\nu+2} \\ \vdots \\ -s_{2\nu} \end{pmatrix} \tag{3.21}$$

Ještě zbývá doplnit, že koeficient λ_0 je vždy jedna. Nyní, když máme lokalizační polynom kompletní, musíme spočítat jeho kořeny.

$$\Lambda(x) = (1 + \alpha^{i_1} x)(1 + \alpha^{i_2} x) \ldots (1 + \alpha^{i_\nu} x) \tag{3.22}$$

Když máme kořeny lokalizačního polynomu 3.22 ve tvaru $\alpha^{i_1}, \alpha^{i_2}, \ldots, \alpha^{i_\nu}$, tak pozice chyb (počítáno od 0 zprava) budou $i_1, i_1, \ldots, i_1$.

Nyní už víme, které bity jsou chybné a tak můžeme provést opravu jejich invertováním. Tím je proces dekódování dokončen.

Pro implementaci bude potřeba počítat generující polynom, minimální polynom. Dále počítat kódovou vzdálenost, permutovat správně sloupce při tvorbě kontolní matice. Tvořit podmatice (vynechání sudých sloupců).

Tabulka 3.4: Tabulka prvků tělesa GF(2^3)

symbol	binární zápis
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Tabulka 3.5: Operace sčítání na GF(2^3)

$\oplus$	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	0	3	2	5	4	7	6
2	2	3	0	1	6	7	4	5
3	3	2	1	0	7	6	5	4
4	4	5	6	7	0	1	2	3
5	5	4	7	6	1	0	3	2
6	6	7	4	5	2	3	0	1
7	7	6	5	4	3	2	1	0

3.8 Reed-Solomonovy kódy

Reed-Solomonovy kódy (dále RS kódy) patří do skupiny tzv MDS kódů (z angličtiny maximum distance separable). MDS kódy mají tu vlastnost, že kvzd = n-k+1. Proto skutečná kódová vzdálenost je rovna minimální kódové vzdálenosti. V RS kódech se opět pracuje s polynomy. Koeficienty polynomů však nejsou z abecedy 0,1, ale jako koeficienty budou prvky konečného tělesa GF(q). Tyto prvky jsou označovány v angličtině jako byte - slabiky. Jedna taková slabika se zkládá z m bitů. Není tedy pravidlem, že byte musí mít 8 bitů [4]. Tyto prvky pro GF(2^3) shrnuje tabulka 3.4. Pozor, kódová vzdálenost se počítá jako počet lišících se symbolů dvou slov. Pracujeme se symboly a tak počet lišících se bitů v binární reprezentaci není podstatný. Dále se také mění označení kódu. Umí-li kód opravit 2 slabiky každou délky 3 bity, kód se označí jako D3EC. Počet opravitelných slabik je roven (kvzd-1)/2 zaokrouhleno dolů. Kódová vzdálenost se rovná jedna plus počet přidaných znaků.

Pro práci s RS kódy budeme potřebovat definovat dva druhy operací. Operace mezi koeficienty polynomů budou definovány podle pravidel počítání na konečných tělesech. Tyto operace pro m=3 a ireducibilní polynom ve tvaru 1011 shrnují tabulky operace sčítání 3.5 a operace násobení 3.6.

Tabulka 3.6: Operace násobení na GF(2^3)

$\odot$	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7
2	2	4	6	3	1	7	5
3	3	6	5	7	4	1	2
4	4	3	7	6	2	5	1
5	5	1	4	2	7	3	6
6	6	7	1	5	3	2	4
7	7	5	2	1	6	4	3

Tabulka 3.7: Zechovy logaritmy

α^i	i
1	0
2	1
3	3
4	2
5	6
6	4
7	5

Tabulka 3.8: Zechovy antilogaritmy

i	α^i
0	1
1	2
2	4
3	3
4	6
5	7
6	5
7	1

Protože násobení s redukcí modulo ireducibilní polynom je poměrně složitá operace, někdy se násobení provádí za pomocí logaritmů a antilogaritmů. Každý prvek konečného tělesa lze vyjádřit jako mocninu generátoru (zde označeného jako α) konečného tělesa. Potom součin prvků α^x a α^y spočítáme jako $\alpha^x \cdot \alpha^y = \alpha^{(x+y)}$. Logaritmy a antilogaritmy jsou uložené v tabulkách. Nazývají se Zechovy logaritmy a antilogaritmy [1]. Pro GF(2^3) s generátorem $\alpha = 010$ a ireducibilním polynomem ve tvaru 1011 bude tabulka logartimů mít tvar 3.7 a antilogaritmy budou vypadat takto 3.8.

Dále bude třeba definovat operace mezi polynomy. Zde to začíná být trochu

nepřehledné, protože koeficienty těchto polynomů jsou z GF(2^3) a tedy jsou také vlastně polynomy. Proto dále budu mluvit o operacích mezi polynomy. Operace mezi koeficienty polynomů byly popsány výše.

Definujme nejprve sčítání polynomů P(x)+Q(x).
$P(x) = p_n\alpha^n + p_{(n-1)}\alpha^{(n-1)} + ... + p_1\alpha^1 + p_0$.
$Q(x) = q_n\alpha^n + q_{(n-1)}\alpha^{(n-1)} + ... + q_1\alpha^1 + q_0$.
$P(x)+Q(x) = (p_n \oplus q_n)\alpha^n + (p_{(n-1)} \oplus q_{(n-1)})\alpha^{(n-1)} + ... + (p_1 \oplus q_1)\alpha^1 + p_0 \oplus q_0$, kde $\oplus$ značí sčítání mezi koeficienty. Je to sčítání prvků konečného tělesa. A protože zde pracujeme s konečnými tělesy 2^m, půjde o operaci bitový xor. Nedochází zde tedy k žádnému přenosu do vyšších řádů.

Násobení polynomů bude probíhat dle distributivního zákona. Jednotlivé členy jednoho polynomu se roznásobí s druhým polynomem. Poté se koeficienty stejných mocnin x posčítají podle pravidel pro sčítání polynomů. Chceme spočítat součin $R(x) = P(x) \cdot Q(x)$, kde $R(x) = r_n\alpha^n + r_{(n-1)}\alpha^{(n-1)} + ... + r_1\alpha^1 + p_0$.
Potom koeficient r_m spočteme podle vzorce 3.23. A protože se jedná o operaci mezi koeficienty, součtem se zde myslí operace xor.

$$r_m = \sum_{i=0}^{m} p_{m-i} * r_i \tag{3.23}$$

Nyní, když už máme popsané základní operace, přejděme k tvorbě generátoru kódu. Pro kódovou vzdálenost δ bude generátor vypadat takto: $G(x) = (x+\alpha) \cdot (x+\alpha^2) \cdots (x+\alpha^\delta - 1)$.

Opět můžeme kódovat pouhým pronásobením zprávy generujícím polynomem. Ale kód by byl nesystematický a dekódovaní by bylo složité. Systematický kód bude generován posunutím o $\delta - 1$ pozic vlevo - což odpovídá násobení polynomem $x^{\delta-1}$. Takto upravenému polynomu se spočítá zbytek po dělení generátorem a tento zbytek se přičte.
$B(x) = A(x) \cdot x^{\delta-1} + A(x) \cdot x^{\delta-1} \% G(x)$.

Příklad 1: Příklad pro D3EC kód s ireducibilním polynomem (1011). Pracujeme se symboly z GF(2^3). Pro 2 opravy je potřeba kódová vzdálenost $\delta = 5$.
$G(x) = (x+\alpha)\cdot(x+\alpha^2)\cdot(x+\alpha^3)\cdot(x+\alpha^4) = x^4+3x^3+x^2+2x+3$.
Generující matice

$$G = \begin{pmatrix} 1 & 0 & 0 & 6 & 1 & 6 & 7 \\ 0 & 1 & 0 & 4 & 1 & 5 & 5 \\ 0 & 0 & 1 & 3 & 1 & 2 & 3 \end{pmatrix} \tag{3.24}$$

Kontrolní matice

$$H = \begin{pmatrix} \alpha^6 & \alpha^5 & \alpha^4 & \alpha^3 & \alpha^2 & \alpha^1 & 1 \\ \alpha^{12} & \alpha^{10} & \alpha^8 & \alpha^6 & \alpha^4 & \alpha^2 & 1 \\ \alpha^{18} & \alpha^{15} & \alpha^{12} & \alpha^9 & \alpha^6 & \alpha^3 & 1 \\ \alpha^{24} & \alpha^{20} & \alpha^{16} & \alpha^{12} & \alpha^8 & \alpha^4 & 1 \end{pmatrix} = \begin{pmatrix} 5 & 7 & 6 & 3 & 4 & 2 & 1 \\ 7 & 3 & 2 & 5 & 6 & 4 & 1 \\ 6 & 2 & 7 & 4 & 5 & 3 & 1 \\ 3 & 5 & 4 & 7 & 2 & 6 & 1 \end{pmatrix} \tag{3.25}$$

Dekódování je o dost složitější. Rozdělíme ho na několik fází.

- Vypočtení syndromu
- Určení počtu chyb
- Určení pozic chyb
- Vypočtení velikosti chyb
- Odečtení chyb - oprava.

Syndrom odpovídá součinu $c \cdot H^T$. Získáme tak $\delta - 1$ symbolů syndromu $s = (s_1, s_2, \cdots, s_{\delta-1})$. Protože u cyklických kódů nepoužíváme kontrolní a generující matici, spočteme s_j tak, že do přijatého slova C(x) dosadíme j-tý generující kořen.

$$s_j = C(\alpha^j) \tag{3.26}$$

Je-li syndrom nulový, můžeme v této fázi skončit. Přenos proběhl bez chyb. V opačném případě musíme dopočítat chyby.

Každé dva znaky syndromu nám umožňují opravit jednu chybu. [1]
Počet opravitelných chyb označme t. Máme tedy 2t symbolů syndromu. Když máme spočteny všechny znaky syndromu, sestrojíme následující matici:

$$\Theta_\nu = \begin{pmatrix} s_1 & s_2 & \cdots & s_\nu \\ s_2 & s_3 & \cdots & s_{\nu+1} \\ \vdots & \vdots & \vdots & \vdots \\ s_\nu & s_{\nu+1} & \cdots & s_{2\nu-1} \end{pmatrix} \tag{3.27}$$

Kde za ν dosadíme t. Dále spočítáme determinant této matice. Je-li determinant nulový, jsou informace z některých symbolů syndromu redundantní a víme, že došlo k méně než ν chybám. Takto se sestrojí menší matice, kde se za ν dosadí $\nu - 1$. Hodnotu ν budeme snižovat dokud nedostaneme nenulový determinant. Až se nám to podaří, bude hodnota ν rovna skutečnému počtu chyb. Nyní, když máme matici Θ_ν s nenulovým determinantem, můžeme udělat inverzní matici ${\Theta_\nu}^{-1}$.

[1] RS kódy lze využít i k opravě nečitelných znaků. Víme-li pozici znaku, který chceme opravit, bude „cena opravy“ pouze jeden symbol syndromu. Tato práce se však tímto typem chyb nezabývá

Dále už můžeme spočítat lokalizační polynom. Lokalizační polynom má tvar

$$\Lambda_x = 1 + \Lambda_1 x + \cdots + \Lambda_{\nu-1} x^{\nu-1} + \Lambda_\nu x^\nu \tag{3.28}$$

kde Λ_1 až Λ_ν spočítáme ze vztahu 3.29.

$$\begin{pmatrix} \lambda_\nu \\ \lambda_{\nu-1} \\ \vdots \\ \lambda_1 \end{pmatrix} = {\Theta_\nu}^{-1} \begin{pmatrix} -s_{\nu+1} \\ -s_{\nu+2} \\ \vdots \\ -s_{2\nu} \end{pmatrix} \tag{3.29}$$

Dále spočítáme kořeny lokalizačního polynomu.

$$\Lambda(\xi) = (1 + \alpha^{i_1}\xi)(1 + \alpha^{i_2}\xi) \dots (1 + \alpha^{i_\nu}\xi) \tag{3.30}$$

Máme-li kořeny ve tvaru $\alpha_1 = \alpha^{i_1}, \alpha_2 = \alpha^{i_2}, \dots, \alpha_\nu = \alpha^{i_\nu}$, potom jejich logaritmy i_1, i_2, …, i_ν udávají pozice chyb počítané zprava od nuly. Tedy $x^{i_1}, x^{i_2}, \dots, x^{i_\nu}$ vyjádřeno polynomem.

Nyní známe pozice, ve kterých došlo k chybě. Ale nevíme velikost chyby. Pro výpočet velikosti chyb nejprve sestrojíme matici

$$\Omega = \begin{pmatrix} \alpha_1 & \alpha_2 & \cdots & \alpha_\nu \\ \alpha_1^2 & \alpha_2^2 & \cdots & \alpha_\nu^2 \\ \vdots & \vdots & \vdots & \vdots \\ \alpha_1^\nu & \alpha_2^\nu & \cdots & \alpha_\nu^\nu \end{pmatrix} \tag{3.31}$$

kde $\alpha_1, \alpha_2, \cdots, \alpha_\nu$ jsou kořeny lokalizačního polynomu3.30. Dále spočítáme inverzi této matice. Velikost chyb ε_1 až ε_ν dopočítáme podle vzorce

$$\begin{pmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_\nu \end{pmatrix} = \Omega^{-1} \begin{pmatrix} s_1 \\ s_2 \\ \vdots \\ s_\nu \end{pmatrix} \tag{3.32}$$

Nyní známe pozice chyb i jejich velikosti. Sestrojíme tedy polynom zanesené chyby E(x). Pro opravu zprávy zbývá už jen chyby na příslušných pozicích odečíst.

$$D(x) = C(x) - E(x). \tag{3.33}$$

3.9 LDPC

Low Density Parity Check. Někdy také pojmenovány po autorovi jako Gallagerovy kódy. Patří mezi lineární blokové kódy. LDPC kódy se kromě řídké kontrolní matice nevyznačují ničím význačným oproti kódům popsaným výše. Jediné, co je definuje, je to, že paritní matice je řídká. Tedy převážnou většinu

prvků kontrolní matice mají nulovou. Proto se LDPC kódy zpravidla vytvářejí od kontrolní matice.

LDPC kódy dělíme na pravidelné a nepravidelné. Pravidelný LDPC kód je definován jako $C_{LDPC}(n, o, v)$, kde n udává délku kódu, o je počet jedniček ve sloupci a v je počet jedniček v řádku. Počet jedniček ve sloupcích i v řádcích je tedy pro kód konstantní. Navíc počet jedniček v řádku $o \geq 3$. Kód s řídkou kontrolní maticí, který nemá počet jedniček ve sloupcích i v řádcích konstantní, nazveme nepravidelný LDPC kód.

Existuje více metod pro vytváření paritní matice. Jednou takovou možností je náhodná konstrukce matic. Při tvorbě pravidelného LDPC je potřeba dodržet následující pravidla:

- Každý sloupec má konstantní počet jedniček o.
- Každý řádek má konstantní počet jedniček v.
- Matice musí být řídká(obsahuje víc 0 než 1).
- Každé dva řádky matice musí být lineárně nezávislé.
- V Tannerových grafech kontrolní matice by nemělo docházet k cyklům délky 4 (co je to cyklus bude vysvětleno dále v této kapitole). Ekvivalentní požadavek je, že v kontrolní matici nebude existovat podmatice o minimálních rozměrech 2 x 2, která by měla znak 1 ve všech čtyřech rozích.

V praxi se však spíše používá pseudonáhodná konstrukce. Například metoda, kterou použil pro konstrukci Gallager[6]: Gallagerova kontrolní matice H se skládá z několika submatic 3.34.

$$H = \begin{pmatrix} H_1 \\ H_2 \\ \vdots \\ H_m \end{pmatrix} \tag{3.34}$$

H_1 je základní matice. Vznikne tak, že v každém sloupci je jedna jednička. Začne se v prvním řádku a pravidelně se zvyšuje číslo řádku. H_1 pro matici o 9 sloupcích a 3 řádcích bude mít tvar 3.35.

$$H_1 = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix} \tag{3.35}$$

Další podmatice vznikají permutací právě H_1. Tím je zaručeno, že výsledná matice H bude mít konstantní počet jedniček ve sloupcích i v řádcích. Při (pseudo)náhodné konstrukci je potřeba dávat pozor na to, aby nevznikaly

Tabulka 3.9: Parametry LDPC kódů tvořených podle EG

s	n	k	min kvzd	o	v	#1/#0 v matici	hustota kódu
2	15	7	5	4	4	0,276	0,467
3	63	37	9	8	8	0,127	0,587
4	255	175	17	16	16	0,063	0,686
5	1023	781	33	32	32	0,031	0,763
6	4095	3367	65	64	64	0,017	0,822
7	16383	14197	129	128	128	0,08	0,867

lineárně závislé řádky. Také zde může docházet k cyklům délky 4, které nejsou žádoucí.

V následucím odstavci a příkladu jsou shrnuty poznatky, které vycházejí z literatury [7] a [8] o možnosti tvorby LDPC kódu pomocí geometrické konstrukce paritní matice.

Jsou zde uvedeny dvě možnosti konstrukce. Eukleidovská geometrie (EG) a projektivní geometrie (PG). Ke zpracování jsem vybral Eukleidovskou geometrii, protože vytváří matice s lichou kódovou vzdáleností. Vychází se zde z Galoisových těles GF. V obou případech nám vznikne cyklický kód. LDPC kód vytvořený pomocí m-dimenzionální Eukleidovské geometrie nad $GF(2^s)$ označme $EG(m, 2^s)$, kde m značí dimenzi kódu a s stupeň kódu. Pro tuto práci budeme počítat s m=2. Potom ostatní parametry budou vypadat takto:

délka kódu $n = 2^{2s} - 1$,
redundance $r = n - k = 3^s - 1$,
počet informačních bitů $k = 2^{2s} - 3^s$
kódová vzdálenost $kvzd = 2^s + 1$
Kompletní seznam parametrů shrnuje tabulka 3.9.

Kódy vyššího stupně mají lepší korekční schopnost (větší kvzd), ale i nižší relativní redundanci, z čehož vyplývá vyšší informační rychlost.

Příklad 2:

Vytvoříme kód EG(2,4): parametry tohoto kódu můžeme vidět v 1. řádku tabulky. Máme m = 2, s = 2. Vycházíme z Galoisova pole $GF(2^{2\cdot2})$, tedy GF(16). Primitivní polynom, pomocí kterého je toto pole generováno, má tvar $p(x) = x^4 + x + 1$. Nyní vyjádříme prvky α^0 až α^{14} ve tvaru zbytku po dělení primitivním polynomem. Toto shrnuje tabulka prvků tělesa GF(16) 3.3. Nyní spočteme vytvářecí vektor v. Ten bude mít délku n a bude obsahovat $2^s = 4$ body. Ty se spočítají jako $\alpha^{14} + \pi \cdot \alpha$. Nejprve musíme nalézt prvek β takový, aby platilo $\beta^{2^s-1} + 1 = 0$. Zde vyhovuje $\beta = \alpha^5$. Nyní spočítáme prvky $\{0,1,\beta,\beta^2\}$(obecně $\{0,1,\beta,\beta^2,\cdots,\beta^{2s-2}\}$). Po dosazení $\beta = \alpha^5$ dostaneme hodnoty $\{0,1,\alpha^5,\alpha^{10}\}$, ty postupně všechny dosadíme za π.

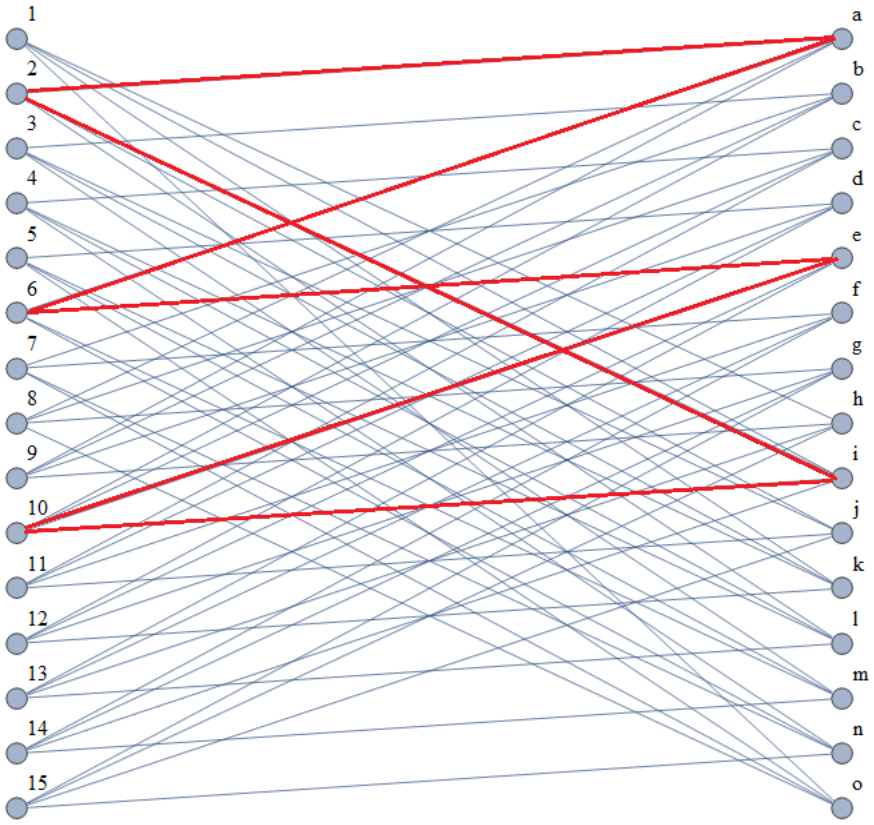
Po každém dosazení získáme jeden prvek.
$\alpha^{14} + 0 \cdot \alpha = \alpha^{14}$
$\alpha^{14} + 1 \cdot \alpha = \alpha^{7}$
$\alpha^{14} + \alpha^{5} \cdot \alpha = \alpha^{8}$
$\alpha^{14} + \alpha^{10} \cdot \alpha = \alpha^{10}$

Ve vektoru v budou jedničky na pozicích odpovídajících mocninám vypočtených prvků. Na ostatních pozicích budou nuly. V tomto případě $v = 100010110000000$. Kontrolní matice se z vektoru vytvoří snadno. Bude to čtvercová matice o rozměru n (délka vektoru v). Na prvním řádku bude vektor v, na $m - tém$ řádku bude vektor v, na nějž se aplikuje cyklický posun vlevo o $m - 1$ pozic, kde $1 \leq m \leq v$. Kontrolní matice pro $v = 100010110000000$ bude tedy vypadat takto:

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

Generující matici je možno vytvořit obvyklým způsobem z kontrolní tak, jak udává rovnice 3.1. Nicméně zde není zaručeno, že dostaneme opět řídkou matici. Výhoda řídké matice spočívá v menším počtu operací a tím vyšší rychlosti výpočtů. Existují i další metody, jak vhodně získat generující matici, aniž by příliš přibylo nenulových prvků. O takovém způsobu se píše například v tomto článku [6] na stranách 17 až 20.

Tannerovy grafy. Je to grafická reprezentace paritní matice. V Tannerově grafu jsou dva typy uzlů - bitové a součtové. Bitových uzlů je tolik, kolik má matice v řádku znaků, počet součtových uzlů je pak roven počtu znaků ve sloupci kontrolní matice. Můžeme říci, že bitových uzlů je n. Ale nelze říci, že součtových uzlů bude k. Kontrolní matice, na rozdíl od generující, může obsahovat i lineárně závislé řádky. Je to kvůli rychlejšímu dekódování. A protože Tannerovy grafy popisují matici nikoli podprostor, i tyto lineárně



Obrázek 3.2: Tannerův graf pro kontrolní matici z příkladu 2

závislé řádky musí být zahrnuty v grafu. V Tannerově grafu každý sloupec matice H představuje jeden součtový uzel a každý řádek představuje jeden bitový uzel. Každá jednička v kontrolní matici představuje hranu Tannerova grafu mezi bitovým a součtovým uzlem příslušného řádku a sloupce. Nula značí, že mezi body hrana není. Lze na to hledět tak, že každý součtový uzel zabezpečuje sudou paritou tu podmnožinu datových bitů, ke kterým z něj vede hrana. Posloupnost hran v Tannerově grafu, která začíná i končí ve stejném bodě, nazveme cyklus. Příklad: Tannerův graf pro matici z příkladu 2 ukazuje obrázek 3.2

Bitové uzly jsou označeny čísly odpovídajícími pořadí řádku matice. Součtové uzly jsou označeny písmeny abecedně od nejnižší mocniny polynomu - pravý sloupec odpovídá součtovému uzlu a, levý pak součtovému uzlu o. Červeně je zvýrazněn cyklus délky 6.
Nyní přejděme k dekódování. Způsob dekódování je asi hlavní rys, který dělá LDPC kódy výjimečnými. Z Tannerových grafů víme, že uzly dělíme na bitové a součtové. Pro úspěšnou rekonstrukci původní zprávy potřebujeme, aby všechny součtové uzly odpovídaly součtu těch bitů, ke kterým jsou připojeny.

Rozlišujeme dva hlavní přístupy k dekódování. Soft-decision a hard-decision. Soft-decision nepracuje pouze se znaky 0 a 1, ale pracuje s pravděpodobnostmi příslušných znaků. V mezikrocích výpočtu tedy máme pravděpodobnosti jednotlivých znaků v jednotlivých uzlech. Když jsme si všemi uzly jisti nad určitou mez, dekódování skončí. Hard-decision má v každém kroku k jednotlivým uzlům přiřazeno buď 0 nebo 1. A právě na hard-decision přístup se tato práce zaměřuje.

Jednotlivé uzly Tannerova grafu komunikují pouze s uzly, se kterými jsou propojeny hranou. Proto je tento dekódovací algoritmus vhodný pro paralelizaci.

Dekódující algoritmus je bitflipping algoritmus. Skládá se z několika kroků. Nejprve každý datový uzel pošle svoji hodnotu (nula nebo jedna) ke všem součtovým uzlům, s kterými je spojen hranou. Potom každý součtový uzel zkontroluje, zda jeho hodnota sedí s hodnotou součtu přijatých hodnot od datových uzlů. Dále součtové uzly, ve kterých nesouhlasí kontrolní součet, informují datové uzly, se kterými jsou spojené hranou. Bitové uzly vyhodnotí, od kolika součtových uzlů dostali informaci, že součet nesedí. Je-li to více než polovina uzlů, se kterými jsou spojeny, invertují svoji hodnotu. Všechny tyto kroky se pak opakují tak dlouho, dokud všechny součty nejsou správné, nebo dokud nedosáhne algoritmus předem stanovený počet iterací.

Příklad 3:

Fungování na jednoduchém příkladu. Mějme přijaté slovo b={0, 1, 1, 0, 0, 1} a kontrolní matici ve tvaru 3.36. Nejprve označíme v matici datové uzly písmeny a až f a součtové uzly čísly 1 až 4.

$$\left(\begin{array}{c|cccccc} & f & e & d & c & b & a \\ \hline 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 2 & 0 & 1 & 1 & 0 & 1 & 0 \\ 3 & 1 & 0 & 0 & 0 & 1 & 1 \\ 4 & 0 & 0 & 1 & 1 & 0 & 1 \end{array}\right)$$

Odpovídající Tannerův graf ukazuje obrázek 3.3.

Krok 1: datové uzly a,d,e=1; b,c,f =0.
uzel 1 sečte f+e+c = 1 tedy nesouhlasí součet. Proto těmto uzlům odešle opačné hodnoty, než které od nich dostal. 1 -> c,f; 0 -> e
uzel 2 sečte e+d+b = 0 tedy součet souhlasí. Proto těmto uzlům odešle stejné hodnoty, jaké od nich dostal. 1 -> e,d; 0 -> b
uzel 3 sečte f+b+a = 1 tedy nesouhlasí součet. Proto těmto uzlům odešle opačné hodnoty, než které od nich dostal. 1 -> f,b; 0 -> a
uzel 4 sečte d+c+a = 0 tedy součet souhlasí. Proto těmto uzlům

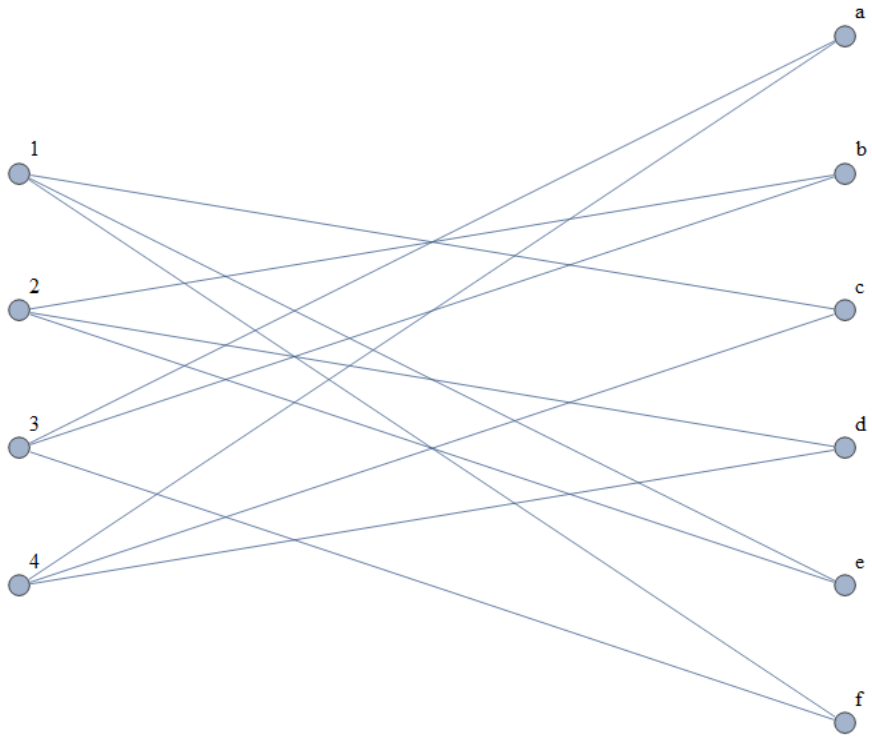
odešle stejné hodnoty, jaké od nich dostal. 1 -> d,a; 0 -> c

Poté, co jsou všechny zprávy součtových uzlů odeslány, přicházejí na řadu datové uzly. Uzel a měl hodnotu 1 a přijal hodnoty 0 a 1 (od uzlů 3 a 4). Majorita je 1
Uzel b měl hodnotu 0 a přijal hodnoty 0 a 1 (od uzlů 2 a 3). Majorita je 0
Uzel c měl hodnotu 0 a přijal hodnoty 1 a 0 (od uzlů 1 a 4). Majorita je 0
Uzel d měl hodnotu 1 a přijal hodnoty 1 a 1 (od uzlů 2 a 4). Majorita je 1
Uzel e měl hodnotu 1 a přijal hodnoty 0 a 1 (od uzlů 1 a 2). Majorita je 1
Uzel f měl hodnotu 0 a přijal hodnoty 1 a 1 (od uzlů 1 a 3). Majorita je 1

Uzel f tedy změnil hodnotu. Proto se pokračuje dál.
Krok 2: datové uzly a,d,e,f=1; b,c =0.
uzel 1 sečte f+e+c = 0 tedy součet souhlasí. Proto těmto uzlům odešle stejné hodnoty, jaké od nich dostal. 1 -> f,e; 0 -> c
uzel 2 sečte e+d+b = 0 tedy součet souhlasí. Proto těmto uzlům odešle stejné hodnoty, jaké od nich dostal. 1 -> e,d; 0 -> b
uzel 3 sečte f+b+a = 0 tedy součet souhlasí. Proto těmto uzlům odešle stejné hodnoty, jaké od nich dostal. 1 -> f,a; 0 -> b
uzel 4 sečte d+c+a = 0 tedy součet souhlasí. Proto těmto uzlům odešle stejné hodnoty, jaké od nich dostal. 1 -> d,a; 0 -> c

Poté, co jsou všechny zprávy součtových uzlů odeslány, přicházejí na řadu datové uzly. Uzel a měl hodnotu 1 a přijal hodnoty 1 a 1 (od uzlů 3 a 4). Majorita je 1
Uzel b měl hodnotu 0 a přijal hodnoty 0 a 0 (od uzlů 2 a 3). Majorita je 0
Uzel c měl hodnotu 0 a přijal hodnoty 0 a 0 (od uzlů 1 a 4). Majorita je 0
Uzel d měl hodnotu 1 a přijal hodnoty 1 a 1 (od uzlů 2 a 4). Majorita je 1
Uzel e měl hodnotu 1 a přijal hodnoty 1 a 1 (od uzlů 1 a 2). Majorita je 1
Uzel f měl hodnotu 1 a přijal hodnoty 1 a 1 (od uzlů 1 a 3). Majorita je 1

Všechny součty sedí. A všem datovým uzlům se vracejí hodnoty,

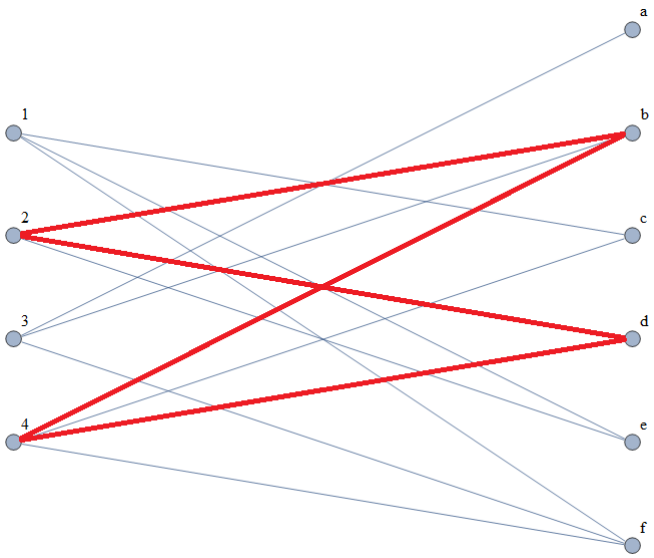


Obrázek 3.3: Tannerův graf pro kontrolní matici 3.36

které odesílají - proto může algoritmus skončit(nic by se neměnilo, kdyby pokračoval). Chyba tedy byla v bitu f. Kódové slovo po opravě tedy vypadá {f, e, d, c, b, a} = {1, 1, 1, 0, 0, 1}.

$$H = \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix} \tag{3.36}$$

Stejný lineární podprostor a tedy i kód může být popsán různými maticemi. Matice H^* 3.37 vznikla z matice H 3.36 přičtením třetího řádku ke čtvrtému. Obě matice tedy popisují stejný kód. Volba, kterou matici použijeme, však může mít vliv na vlastnosti kódu. Matice H^* obsahuje dva cykly délky 4. Jeden z nich je v matici zvýrazněn tučně. Že dekódování stejného slova s touto maticí nebude fungovat, bude předvedeno v kapitole Testování



Obrázek 3.4: Tannerův graf pro kontrolní matici 3.37

v příkladu 57.

$$H^* = \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & \mathbf{1} & 0 & \mathbf{1} & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & \mathbf{1} & 1 & \mathbf{1} & 0 \end{pmatrix} \tag{3.37}$$

3.10 Prokládané kódy

Chceme-li se chránit proti shluku chyb délky m, připravíme si matici A o m řádcích. Do ní nejprve po sloupcích vepíšeme přenášená data (zatím bez redundance, tedy vektor a).
Příklad pro $a = (a_1, a_2, \ldots, a_{12})$:

$$\begin{pmatrix} a_1 & a_4 & a_7 & a_{10} \\ a_2 & a_5 & a_8 & a_{11} \\ a_3 & a_6 & a_9 & a_{12} \end{pmatrix}$$

Dojde-li k shluku chyb délky menší či rovné m , budeme mít zaručeno, že na každém řádku bude z tohoto shluku maximálně jedna chyba. Nyní se tedy z problému shluku chyb stává problém zabezpečení jedné chyby na jednom řádku. Vezmeme tedy řádek po řádku a každý zvlášť zakódujeme zatím obecně kódem K:

$$\begin{pmatrix} b_1 & b_4 & b_7 & b_{10} & b_{13} & b_{16} & b_{19} \\ b_2 & b_5 & b_8 & b_{11} & b_{14} & b_{17} & b_{20} \\ b_3 & b_6 & b_9 & b_{12} & b_{15} & b_{18} & b_{21} \end{pmatrix}$$

Prokládaný kód K*(n*,k*) nám vznikne z kódu K(n,k) aplikovaném na m řádků. Parametry takového kódu budou:
$n^* = n \cdot m$,
$k^* = k \cdot m$.

Od možností kódu K se odvíjejí možnosti kódu K*. Pokud K detekuje jednu chybu, pak K* detekuje jeden shluk do délky m. Pokud K umožňuje opravit jednu chybu, pak K* opravuje jeden shluk do délky m. Pokud K umožňuje opravit x chyb, pak K* opravuje x shluků do délky m. Protože však mohou tyto shluky být hned za sebou, je možné opravit jeden shluk délky $x \cdot m$. Jde o to, aby žádný řádek nebyl zasažen více než x krát.

Příklad pro opravu shluku chyb do délky m=3:
a = 111 001 000 100
nyní vepíšeme do matice A o m řádcích:

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

Jako K zvolíme Hammingův kód K(7,4) s Generující maticí G=

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

Zabezpečíme postupně řádky matice A kódem K:
$B = A \cdot G =$

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

Vektor b vznikne přečtením matice B po sloupcích.
Tedy b = 111 001 000 100 110 101 010. A můžeme si všimnout, že když K byl systematický kód, $K*$ je také systematický.

Zvolíme-li jako K kód, který má generátor $G(x)$, generátor kódu $K*$ bude $G*(x) = G(x^m)$. Podobně potom platí $H*(x) = H(x^m)$

Kapitola 4

Návrh řešení

4.1 Základní funkce společné pro většinu kódů

Nejprve uvedu několik základních funkcí, které jsou potřebné pro práci s blokovými kódy.

Všechny kódy, se kterými v této práci pracujeme, lze popsat ve tvaru kontrolní a generující matice.

Takže základní funkcí bude samotné zakódování jednoho bloku pomocí generující matice. To se provede jako násobení matice vektorem. Řekněme, že toto by mohla dělat funkce Encode s vstupem matice a vektoru délky k a výstupem ve formátu vektoru délky n.

Pro spočtení více bloků najednou, případně pro prokládané kódy, přetížím funkci encode tak, aby jako vstup brala místo vektoru matici o stejném rozměru a tím počítala více slov najednou.

Určení syndromu provedeme funkcí decode. Ta by měla jako vstupní parametry kontrolní matici a kódové slovo. Výstupem pak bude syndrom.

Interpretace syndromu a případná následná oprava už bude záležet na konkrétním kódu. Pro maticově generované kódy schopné opravy jedné chyby by měla existovat funkce syndtable, která vypíše tabulku syndromů. Jako vstup bude generující matice.

Pro tvorbu generující matice z kontrolní lze využít již ve Wolframu obsažené funkce NullSpace. Funkce se musí mírně upravit kvůli rozdílným notacím: G Wolfram, Wiki notace

$$\left(\begin{array}{ccc|ccc} c_{1r} & \cdots & c_{11} & 0 & \cdots & 0 & 1 \\ c_{2r} & \cdots & c_{21} & 0 & \cdots & 1 & 0 \\ \vdots & \vdots & \vdots & \vdots & \iddots & \vdots & \vdots \\ c_{kr} & \cdots & c_{k1} & 1 & \cdots & 0 & 0 \end{array}\right)$$

H Wolfram

$$\left(\begin{array}{cccc|ccc} 0 & \cdots & 0 & 1 & c_{k1} & \cdots & c_{11} \\ 0 & \cdots & 1 & 0 & c_{k2} & \cdots & c_{12} \\ \vdots & \iddots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \cdots & 0 & 0 & c_{kr} & \cdots & c_{1r} \end{array}\right)$$

G školní notace

$$\left(\begin{array}{cccc|ccc} 1 & 0 & \cdots & 0 & c_{11} & \cdots & c_{1r} \\ 0 & 1 & \cdots & 0 & c_{21} & \cdots & c_{2r} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & c_{k1} & \cdots & c_{kr} \end{array}\right)$$

H školní notace

$$\left(\begin{array}{ccc|cccc} c_{11} & \cdots & c_{k1} & 1 & 0 & \cdots & 0 \\ c_{12} & \cdots & c_{k2} & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ c_{1r} & \cdots & c_{kr} & 0 & 0 & \cdots & 1 \end{array}\right)$$

V MATLABu je to podobně jako ve školní notaci, jen G a H jsou obráceně (jednotková matice je z druhé strany).

Funkce na tvorbu kontrolní matice gen2par vrátí nulový prostor a pokusí se upravit matici do tvaru H(školní notace).

Pro generující matici se použije par2gen, která vrátí nulový prostor paritní matice s tím, že bude preferovat tvar G (školní notace).

4.2 Návrh řešení kódu parita

Tvorba paritní matice proběhne tak, že se vytvoří vektor, který bude mít n jedničkových symbolů. Generující matici pak vytvoříme z kontrolní pomocí funkce par2gen. Pro kódování se použije kódovací funkce encode[]. Interpretace přijatého slova proběhne tak, že se oddělí poslední znak. Pokud je tento znak jednička, zpráva není v pořádku a přenos je potřeba opakovat. V případě, že oddělený znak je nula, zbytek vektoru představuje dekódovanou zprávu.

4.3 Návrh řešení Hammingových kódů

Kontrolní matice Hammingova kódu má n sloupců. Každý sloupec je jiný než ostatní a žádný není nulový. Navíc neexistuje jiný sloupec stejné délky. Takže každý sloupec představuje binární zápis jednoho z čísel od 1 do n. Tvorbu matice lze tedy provést tak, že se vezmou čísla od 1 do n a postupně se do sloupců vepíše jejich binární reprezentace. Abychom dostali systematický kód, sloupce s reprezentací čísel, která jsou mocninami dvou, zařadíme až za ostatní

a to v sestupném pořadí mocnin. Funkce, která vytvoří Hammingův kód se bude jmenovat hammgen[]. Jako vstupní parametr bude potřeba předat délku kódu n.

Generující matici získáme z kontrolní pomocí funkce par2gen[]. Tabulku snydromů vytvoříme pomocí funkce syndtable[]. Ke kódování se použije funkce encode[]. Syndrom se spočítá z přijaté zprávy pomocí funkce decode[].

Oprava zprávy v případě nenulového syndromu proběhne tak, že se vyhledá syndrom v tabulce syndromů. Příslušný řádek nám určí přesný bit, kde došlo k chybě. Tento řádek se přičte k přijatému slovu a tím získáme slovo bez chyb. Interpretace zprávy z opraveného přijatého slova proběhne tak, že se vezme prvních k bitů.

4.4 Návrh řešení BCH kódů

Na binární BCH kód lze nahlížet obdobně jako na RS-kód s tím, že koeficienty budou „polynomy“ stupně nula. Tedy 0 nebo 1. Pak lze použít pro oba kódy většinu funkcí stejných. Nicméně vyjádřit slovo $a = x^3 + 1$ jako $\{\{1\}, \{0\}, \{0\}, \{1\}\}$ mi nepřišlo vhodné, proto bude potřeba přidat nějaké části kódu na konverzi z a do tvaru $\{1, 0, 0, 1\}$. Dále funkce pro opravu chyb bude jiná. Není totiž potřeba počítat velikost chyby. Navíc kvůli rozdílným notacím polynomů (seznam pro BCH oproti seznam seznamů pro RS) bude jednodušší provést opravu pomocí vlastní funkce.

4.5 Návrh řešení RS kódy

Kód máme definován trojicí n,r a generujícím prvkem α. Z těchto tří hodnot se spočítá generátor kódu. Před samotným výpočtem kódování a dekódování se připraví tabulka logaritmů a antilogaritmů.

Kódování proběhne vynásobením zprávy polynomem x^r. Následně se tento součin vydělí generátorem a o tuto hodnotu zvýší. Nyní je výsledek dělitelný beze zbytku generátorem. Tím je proces kódování hotov.

Pro určení syndromů se použije funkce, která umí do polynomu dosadit za x příslušné mocniny generujícího prvku α.

Pro nenulové syndromy potřebujeme funkci, která ze syndromů spočítá lokalizační polynom. Dále bude existovat funkce na výpočet matice Θ_ν, tak jak je popsaná zde 3.27. Dále bude potřeba funkce na výpočet inverzní matice s prvky z GF. Součástí bude i funkce počítající determinant takovéto matice. Kořeny lokalizačního polynomu bude hledat funkce hrubou silou. Pro výpočet velikosti chyby bude ještě potřeba nadefinovat vyplnění matice Ω.

4.6 Návrh řešení LDPC kódy

Pro tvorbu LDPC kódů se použije návrh kontrolní matice podle EG. Bude potřeba funkce, která sestrojí vektor v, tedy první řádek matice. Zbylé řádky se vytvoří pomocí funkce RotateLeft. Pro sestrojení vektoru v nejprve funkce FindBeta nalezne prvek β, který splňuje rovnici $\beta^{2^s-1}+1=0$ z příkladu číslo 2. Ten se nalezne hrubou silou postupným dosazováním mocnin α. Pro tvorbu generující matice z kontrolní se použije obecná funkce společná pro většinu kódů.

Dále bude potřeba funkce simulující bitflipping algoritmus. Ta dostane na vstup kontrolní matici a přijatou zprávu. V průběhu dekódování bude vypisovat, co který uzel dělá. Výstupem bude buď zcela nebo částečně opravené kódové slovo.

4.7 Návrh řešení prokládané kódy

Pro tvorbu generátoru potřebujeme zadat hodnoty m a $G(x)$, kde m je délka shluku a $G(x)$ je generátor prokládaného kódu. Protože platí $G^*(x) = G(x^m)$, můžeme vytvořit generátor jednoduchým umocněním. Mocniny polynomu s binárními koeficienty lze tvořit jednoduše. Chceme-li vytvořit m-tou mocninu polynomu, mezi každé dva znaky vektorového zápisu vložíme $(m-1)$ znaků nula.

Kontrolní polynom lze získat běžným způsobem $H^*(x) = (x^n+1)/G^*(x)$ nebo podobně z kontrolní matice původního kódu jako $H^*(x) = H(x^m)$.

Kapitola 5

Řešení

5.1 Použité datové typy/reprezentace polynomů a prvků GF

Pro uchování koeficientů polynomů se použije seznam - fuknce list[]. Do seznamu se ukládají koeficienty od nejvyšších mocnin po nejnižší. Prvky tělesa GF jsou také uloženy jako sekvence 0 a 1 v seznamu. Prvek sám o sobě si nepamatuje, z jakého je tělesa. Generátor tedy musí být přenášen spolu s prvkem GF.

5.2 Převody mezi formáty

5.2.1 vec2pol

Funkce ze seznamu koeficientů $\{c_0, c_1, c_2, \dots\}$ polynomu složí polynom ve tvaru $c_0x^0 + c_1x^1 + c_2x^2 + \dots$.

Příklad 4:

In: vec2pol[{1,0,0,1,1}]
Out: $x^4 + x + 1$

5.2.2 pol2vec

Funkce vezme polynom a uloží jeho koeficienty do seznamu. Koeficienty jsou uloženy podle mocnin sestupně.

Příklad 5:

In: pol2vec[$x^4 + x + 1$]
Out: {1,0,0,1,1}

5.2.3 TraditionalForm

Funkce z knihovny Wolfram. Vyjádří polynom ve tvaru $c_n x^n + \cdots + c_2 x^2 + c_1 x^1 + c_0 x^0$. Funguje pro koeficienty c ze Z_2.

Příklad 6:

In: TraditionalForm[$1 + x + x^4$]
Out: $x^4 + x + 1$

5.2.4 HexForm

Seznam vektorů binárních čísel převede na seznam těchto čísel vyjádřených v haxadecimálním tvaru. Výstup je určen pouze pro zobrazení. Další fungování výpočtů v tomto tvaru není podporováno.

Příklad 7:

In: a = {{1, 0, 0}, {1, 1, 1}, {1, 0, 0}}
In: HexForm[a];
Out: {4, 7, 4}

5.2.5 UnhexForm

Funkce přijme seznam hexadecimálních čísel v textovém tvaru (důležité je obalit čísla uvozovkami i v případě, že se jedná o cifry 0-9). Druhým parametrem je l udávající délku binárního zápisu výstupních čísel. Jsou-li v seznamu prvky tělesa GF, l bude stupeň ireducibilního polynomu. Funkce převede hexadecimální čísla do formátu vhodného pro kódování. Výstupem funkce je seznam, který má jako prvky seznamy cifer binárního zápisu vstupnch prvků.

Příklad 8:

In: a = {"4", "7", "4"}
In: UnhexForm[a, Length[IP] - 1];
Out: {{1, 0, 0}, {1, 1, 1}, {1, 0, 0}}

5.3 Kódy generované maticemi

5.3.1 Kódování maticových kódů

Funkce encode má na vstupu argumenty matice G a vektor v. Funkce provede maticové násobení maticí vektorem. Výstupem funkce je výsledek tohoto násobení.

Příklad 9:

In: G={ {1,0,0,1}, {0,1,0,1}, {0,0,1,1} };
In: G // MatrixForm
Out:

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

In: v={1,0,1};
In: encode[G,v]
Out: {1,0,1,0}

5.3.2 Výpočet syndromu maticových kódů

Funkce decode má na vstupu argumenty matice H a vektor v. Funkce provede maticové násobení $v \cdot H^T$. Vynásobením vznikne syndrom. Ten je výstupem této funkce.

Příklad 10:

In: H={
{1,1,0},
{1,0,1},
};
In: H//MatrixForm
Out:

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

In: v={1,1,1};
In: decode[G,v]
Out: {0,0}

5.3.3 Převrácení pořadí sloupců

Funkce DescCol přijme na vstupu matici. Matici transponuje. Dále pomocí funkce Reverse převrátí pořadí řádků. Nakonec matici opět transponuje.

Příklad 11:

In: H = hammgen[3];
In: H // MatrixForm
Out:

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

In: DescCol[H] // MatrixForm
Out:

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

5.3.4 Tvorba generující matice z paritní

Funkce par2gen přijme na vstupu paritní matici. Té seřadí pozpátku sloupce pomocí funkce DescCol. Pomocí funkce NullSpace se vytvoří nulový prostor. Nakonec se zase pomocí funkce DescCol sloupce vrátí.

Příklad 12:

In: H = hammgen[3];
In: H//MatrixForm
Out:

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

In: G = par2gen[H] // MatrixForm
Out:

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

5.3.5 Tvorba paritní matice z generující

Funkce gen2par přijme na vstupu generující matici. Té seřadí pozpátku řádky pomocí funkce Reverse. Pomocí funkce NullSpace se vytvoří nulový prostor. Nakonec se zase pomocí funkce Reverse řádky vrátí.

Příklad 13:

In: G = par2gen[hammgen[3]] // MatrixForm
Out:

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

In: H = gen2par[G] // MatrixForm
Out:

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

5.3.6 Oříznutí kontrolních bitů

Funkce Unencode přijme kódové slovo a počet paritních bitů. Z konce kódového slova počet paritních bitů odebere a vrátí data odpovídající kódovému slovu.

Příklad 14:

In: Unencode[{1, 0, 0, 1, 1, 0, 0},3]
Out: {1, 0, 0, 1}

5.3.7 Tvorba systematické kontrolní matice pro Hammingovy kódy „stupně“ r

Funkce přijme argument r a spočítá $n = 2^r - 1$. Kontrolní matice Hammingova kódu má n sloupců. Každý sloupec představuje binární zápis jednoho z čísel od 1 do n. Kvůli snadnější implementaci se matice vytvoří po řádcích a poté se transponuje. Algoritmus nejprve vytvoří binární podobu čísla, převede ji na vektor. Tyto vektory vkládá do připravených seznamů. Jedná-li se o mocninu dvou, připojí vektor na začátek jednoho seznamu, v opačném případě ji připojí na začátek druhého seznamu. Poté se seznamy sloučí tak, že se první připojí za druhý pomocí funkce join. Nakonec se ještě provede zmiňovaná transpozice. Na získání G z H, když H je v systematickém tvaru, už máme funkci par2gen(H)5.3.4.

Příklad 15:

In: H = hammgen[3];
In: H // MatrixForm
Out:

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

5.4 Kódy generované polynomem

5.4.1 Násobení polynomů nad Z_2 ve vektorovém zápisu

Oba vektory se pomocí funkce vec2pol převedou na polynomy. Polynomy se mezi sebou vynásobí. Dále pomocí PolynomialMod() se koeficienty mezivýsledku zmodulí dvěma. Nakonec proběhne zpětný převod polynomu do vektorové reprezentace pomocí funkce pol2vec.

Příklad 16:

In: Px = {1,0,0,1};
In: Qx = {1,0,1};

In: PolynomialMul[Px,Qx]
Out: {1,0,1,1,0,1}

5.4.2 Vytvoření kontrolního polynomu z generujícího

Kontrolní polynom se vytvoří vydělením $(x^n + 1)/G(x)$.

Příklad 17:

In: Gx = {1,0,1,1};
In: n=7
In: Gx2Hx[Gx,n]
Out: {1,0,1,1,1}

5.4.3 Zbytek po dělení polynomu polynomem nad Z_2

Oba vektory se pomocí funkce vec2pol převedou na polynomy. Zbytek po dělení polynomem se spočítá pomocí PolynomialMod(). Výsledek se převede pomocí pol2vec zpátky na vektorový zápis.

Příklad 18:

In: Px = {1,1,0,0,1};
In: Qx = {1,1,1};
In: VectMod[Px,Qx]
Out: {1,0}

5.4.4 Doplnění vektoru zleva nulami

Funkce prependZeros přijme vektor a požadovanou délku. Před vektor připojuje nuly, dokud vektor nemá požadovanou délku.

Příklad 19:

In: px = {1,0,1};
In: length = 7;
In: prependZeros[Px,7]
Out: {0,0,0,0,1,0,1}

5.4.5 Sčítání a odčítání prvků GF

Sčítání i odčítání je stejné a provede se pomocí funkce BitXor.

Příklad 20:

In: a = {1,0,0,1};
In: b = {1,0,1,0};
In: GFAdd[a,b]
Out: {0,0,1,1}

5.4.6 Tvorba tabulky antilogaritmů

Funkce přijme generátor a ireducibilní polynom. Postupně tvoří mocniny generátoru, které se, je-li to potřeba, ještě zmodulí ireducubilním polynomem. Prvky se ukládají do seznamu. Algoritmus skončí, když se prvky začnou opakovat. Výstupem funkce je právě seznam prvků - seznam mocnin generátoru.

Příklad 21:

In: alpha = {0,1,0};
In: IP = {1,0,1,1};
In: ALT=AntiLogTable[alpha,IP]
Out: { {0,1,0}, {1,0,0}, {0,1,1}, {1,1,0}, {1,1,1}, {1,0,1}, {0,0,1} }

5.4.7 Antilogaritmus prvku GF

Funkce GFExp[]dostane na vstup tabulku antilogaritmů a exponent. Z tabulky antilogaritmů vyhledá prvek na pozici odpovídající exponentu.

Příklad 22:

In: alpha = {0,1,0};
In: IP = {1,0,1,1};
In: ALT=AntiLogTable[alpha,IP]
Out: { {0,1,0}, {1,0,0}, {0,1,1}, {1,1,0}, {1,1,1}, {1,0,1}, {0,0,1} }
In: AntiLog[ALT,3]
Out: {0,1,1}

5.4.8 Logaritmus prvku GF

Funkce GFLog[] dostane na vstup tabulku antilogaritmů a prvek, který chceme zlogaritmovat. Z tabulky antilogaritmů se nalezne index logaritmovaného prvku. To se provede pomocí funkce Position[]. Tento index odpovídá logaritmu a je výstupem této funkce.

Příklad 23:

In: alpha = {0,1,0};
In: IP = {1,0,1,1};
In: ALT=AntiLogTable[alpha,IP]
Out: { {0,1,0}, {1,0,0}, {0,1,1}, {1,1,0}, {1,1,1}, {1,0,1}, {0,0,1} }
In: Log[ALT,{0,1,1}]
Out: 3

5.4.9 Násobení prvků GF

Funkce GFMul[] přijme na vstupu tabulku antilogaritmů a dva součinitele A a B. Součin A*B se provede jako Exp(Log(A)+Log(B)) za použití funkcí GFLog[] a GFExp[].

Příklad 24:

In: alpha = {0,1,0};
In: IP = {1,0,1,1};
In: ALT=AntiLogTable[alpha,IP]
Out: { {0,1,0}, {1,0,0}, {0,1,1}, {1,1,0}, {1,1,1}, {1,0,1}, {0,0,1} }
In: GFMul[ALT,{0,1,1},{1,0,1}]
Out: {1,0,0}

5.4.10 Dělení prvků GF

Funkce GFDiv[] bude mít na vstupu tabulku antilogaritmů, dělenec A a dělitel B. Podíl A/B se provede jako Exp(Log(A)-Log(B)) MOD počet prvků v tabulce. Využije se funkcí GFLog[] a GFExp[].

Příklad 25:

In: alpha = {0,1,0};
In: IP = {1,0,1,1};
In: ALT=AntiLogTable[alpha,IP]
Out: { {0,1,0}, {1,0,0}, {0,1,1}, {1,1,0}, {1,1,1}, {1,0,1}, {0,0,1} }
In: GFDiv[ALT,{1,0,0},{1,0,1}]
Out: {0,1,1}

5.4.11 Umocnění prvku GF

Funkce GFPow dostane jako vstupní parametry tabulku antilogaritmů, mocněný prvek p a exponent e. Výsledek se spočítá jako Exp(Log(p)*e) MOD počet prvků v tabulce antilogaritmů. Tento výsledek je návratovou hodnotou funkce.

Příklad 26:

In: alpha = {0,1,0};
In: IP = {1,0,1,1};
In: ALT=AntiLogTable[alpha,IP]
Out: { {0,1,0}, {1,0,0}, {0,1,1}, {1,1,0}, {1,1,1}, {1,0,1}, {0,0,1} }
In: GFPow[ALT,{0,1,1},3]
Out: {1,0,0}

5.4.12 Inverze prvku GF

Funkce GFInv dostane na vstup tabulku antilogaritmů a prvek p. Inverzi k prvku p spočítá jako Exp(r-Log(p)), kde r je počet prvků v tabulce antilogaritmů (řád tělesa).

Příklad 27:

In: alpha = {0,1,0};
In: IP = {1,0,1,1};
In: ALT=AntiLogTable[alpha,IP]
Out: { {0,1,0}, {1,0,0}, {0,1,1}, {1,1,0}, {1,1,1}, {1,0,1}, {0,0,1} }
In: GFInv[ALT,{0,1,1},3]
Out: {1,1,0}

5.4.13 Polynomy nad GF - násobení skalárem

Funkce GFPolyScale přijímá jako vstupní parametry tabulku antilogaritmů, polynom ve formě vektoru koeficientů a prvek tělesa GF, kterým má být polynom vynásoben. Ve for cyklu jsou všechny koeficienty polynomu pronásobeny prvkem tělesa GF pomocí funkce GFmul.

Příklad 28:

In: alpha = {0,1,0};
In: IP = {1,0,1,1};
In: ALT=AntiLogTable[alpha,IP]
Out: { {0,1,0}, {1,0,0}, {0,1,1}, {1,1,0}, {1,1,1}, {1,0,1}, {0,0,1} }
In: Px={{0,1,0},{1,0,0},{0,1,1}};
In: c = {1,1,1};
In: GFPolyScale[ALT,Px,c]
Out: {{1,0,1},{0,0,1},{0,1,0}}

5.4.14 Polynomy nad GF - sčítání polynomů

Funkce GFPolyAdd přijme dva polynomy ve zkráceném zápisu. Koeficienty stejných mocnin sečte pomocí GFAdd a vrátí výsledek.

Příklad 29:

In: Px={{0,1,0},{1,0,0},{0,1,1}};
In: Qx={{1,0,1},{0,0,1},{0,1,0}};
In: GFPolyAdd[Px,Qx]
Out: {{1,1,1},{1,0,1},{0,0,1}}

5.4.15 Polynomy nad GF - násobení polynomu polynomem

Funkce GFPolyMul přijme tabulku antilogaritmů a dva polynomy ve zkráceném zápisu. Ve dvou zanořených for cyklech vynásobí každý koeficient jednoho polynomu s každým koeficientem druhého polynomu. Vrátí součet těchto součinů jako jeden polynom.

Příklad 30:

In: alpha = {0,1,0};
In: IP = {1,0,1,1};

In: ALT=AntiLogTable[alpha,IP]
Out: { {0,1,0}, {1,0,0}, {0,1,1}, {1,1,0}, {1,1,1}, {1,0,1}, {0,0,1} }
In: Px={{0,1,0},{1,0,0},{0,1,1}};
In: Qx={{1,0,1},{0,0,1},{0,1,0}};
In: GFPolyMul[ALT,Px,Qx]
Out: {{0,0,1},{0,0,0},{1,0,0},{0,0,0},{1,1,0}}

5.4.16 Polynomy nad GF - dělení polynomu polynomem

Vstupem funkce je tabulka antilogaritmů, dělenec a dělitel ve formě zkráceného zápisu polynomu. Výstupem funkce je seznam, v němž je podíl a zbytek po dělení.

Příklad 31:

In: alpha = {0,1,0};
In: IP = {1,0,1,1};
In: ALT=AntiLogTable[alpha,IP]
Out: { {0,1,0}, {1,0,0}, {0,1,1}, {1,1,0}, {1,1,1}, {1,0,1}, {0,0,1} }
In: Px={{0,1,0},{1,0,0},{0,1,1}};
In: Qx={{1,0,1},{0,0,1},{0,1,0}};
In: {quotient, remainder} = GFPolyDiv[ALT,Px,Qx]
Out: {{{0,1,0}},{{1,1,0},{1,1,1}}}
In: quotient
Out: {{0,1,0}}
In: remainder
Out: {{1,1,0},{1,1,1}}

5.4.17 Polynomy nad GF - výpočet hodnoty polynomu po dosazení za x

Vstupem je tabulka antilogaritmů, hodnota x a polynom ve zkráceném tvaru. Výstup se spočítá dle Hornerova schématu (od koeficientů u nejvyšších mocnin). Výsledkem je prvek GF odpovídající dosazení hodnoty prvku x do polynomu.

Příklad 32:

In: alpha = {0,1,0};
In: IP = {1,0,1,1};
In: ALT = AntiLogTable[alpha,IP]
Out: { {0,1,0}, {1,0,0}, {0,1,1}, {1,1,0}, {1,1,1}, {1,0,1}, {0,0,1} }
In: Px = {{0,1,0},{1,0,0},{0,1,1}};
In: x = {1,0,1};
In: GFPolyEval[Px,x]

Out: {1,0,0}

5.4.18 Tvorba generátoru pro RS

Vstupem je tabulka antilogaritmů a počet redundantích znaků = $\delta - 1$. Za použití GFPolyMul se spočítá součin $G(x) = (x+\alpha)\cdot(x+\alpha^2)\cdots(x+\alpha^{\delta}-1)$. Hodnota α se nalezne v tabulce antilogaritmů pod indexem 1. Tento součin je výstupem funkce.

Příklad 33:

In: alpha = {0,1,0};
In: IP = {1,0,1,1};
In: ALT = AntiLogTable[alpha,IP];
In: RSGeneratorPoly[ALT, 4]
Out: {{0, 0, 1}, {0, 1, 1}, {0, 0, 1}, {0, 1, 0}, {0, 1, 1}}

5.4.19 Výpočet generující a kontrolní matice pro RS kód

Funkce GFromALT a HFromALT přijímají tabulku antilogaritmů, počet redundantních znaků r a jako třetí parametr je maximální možná délka zprávy $n = 2^s - 1$, kde s je stupeň ireducibilního polynomu. Parametr n lze spočítat jako počet prvků v tabulce antilogaritmů a nemusí být tedy vstupním parametrem.

Příklad 34:

In: IP = {1, 0, 1, 1};
In: ALT = AntiLogTable[0, 1, 0, IP];
In: n = 7;
In: r = 4;
In: G = GFromALT[ALT, r, n] // MatrixForm
Out:

$$\begin{pmatrix} 1 & 0 & 0 & 6 & 1 & 6 & 7 \\ 0 & 1 & 0 & 4 & 1 & 5 & 5 \\ 0 & 0 & 1 & 3 & 1 & 2 & 3 \end{pmatrix}$$

In: H = HFromALT[ALT, r, n] // MatrixForm
Out:

$$\begin{pmatrix} 5 & 7 & 6 & 3 & 4 & 2 & 1 \\ 7 & 3 & 2 & 5 & 6 & 4 & 1 \\ 6 & 2 & 7 & 4 & 5 & 3 & 1 \\ 3 & 5 & 4 & 7 & 2 & 6 & 1 \end{pmatrix}$$

5.4.20 Výpočet minimálních polynomů

Funkce minpol přijme tabulku antilogaritmů a číslo minimálního polynomu. Jako výstup funkce je minimální polynom ve tvaru vektoru.

Příklad níže postupně počítá $M_{\#1}(x), M_{\#3}(x), M_{\#5}(x)$ a $M_{\#7}(x)$.

Příklad 35: In: alpha = {0,0,1,0};
In: IP = {1,0,0,1,1};
In: ALT = AntiLogTable[alpha,IP];
In: MinPol[ALT, 1]
Out: {1, 0, 0, 1, 1}
In:MinPol[ALT, 3]
Out: {1, 1, 1, 1, 1}
In:MinPol[ALT, 5]
Out: {1, 1, 1}
In:MinPol[ALT, 7]
Out: {1, 1, 0, 0, 1}

5.4.21 Tvorba generátoru pro binární kódy BCH

Funkce přijme jako argumenty generující prvek a počet chyb, které chceme opravovat. Nejprve se spočítají minimální mnohočleny pomocí funkce minpol. Ty se mezi sebou pronásobí pomocí funkce PolynomialMul a tím je vytvořen generující polynom. Ten je výstupem této funkce.

Příklad 36:

In: alpha = {0,0,1,0};
In: IP = {1,0,0,1,1};
In: ALT = AntiLogTable[alpha,IP];
In:= BCHGeneratorPoly[ALT, 5]
Out: {1, 1, 1, 0, 1, 0, 0, 0, 1}

5.4.22 Zakódování zprávy pomocí BCH

Funkce SysCode přijme generátor slovo a a generátor g. Pomocí posunu slova a a dopočítání zbytku po dělení generátorem g zakóduje slovo. To je výstupem této funkce. Všechny vstupní i výstupní formáty jsou formou vektoru.

Příklad 37:

In: g = {1, 1, 1, 0, 1, 0, 0, 0, 1};
In: a = 1, 0, 0, 0, 0, 1, 0;
In: b = SysCode[a, g]
Out: {1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1}

5.4.23 Zakódování zprávy pomocí RS

Vstupem funkce je tabulka antilogaritmů, zpráva a počet redundantních znaků δ. Funkce spočte generátor pomocí funkce RSGeneratorPoly. Poté funkce doplní zprávu o nulové koficienty kterých je $\delta - 1$ = stupeň generátoru-1. To udělá pomocí násobení polynomu polynomem. Poté dopočítá zbytek po dělení rozšířené zprávy generátorem. Tento výsledek připojí pomocí funkce Join za původní zprávu a vrátí na výstup.

Příklad 38:

In: alpha = {0,1,0};
In: IP = {1,0,1,1};
In: ALT = AntiLogTable[alpha,IP]
Out: { {0,1,0}, {1,0,0}, {0,1,1}, {1,1,0}, {1,1,1}, {1,0,1}, {0,0,1} }
In: b = RSEncodeMsg[ALT, {{1, 0, 0}, {1, 1, 1}, {1, 0, 0}}, 4]
Out: {{1, 0, 0}, {1, 1, 1}, {1, 0, 0}, {0, 1, 1}, {1, 1, 1}, {0, 0, 0}, {0, 0, 0}}

5.4.24 Spočítání syndromů pro RS/BCH

Vstupem funkce je tabulka antilogaritmů, přijatá zpráva a parametr udávající počet redundantních znaků který je roven $\delta - 1$. Za použití funkce GFPolyEval dosadí postupně kořeny generujícího polynomu. (Ty jsou jednoznačně určeny z tabulky antilogaritmů a parametru delta.) Výsledky dosazených kořenů jsou syndromy. Tyto syndromy jsou výstupem této funkce v podobě seznamu.

Příklad 39:

In: alpha = {0,1,0};
In: IP = {1,0,1,1};
In: ALT = AntiLogTable[alpha,IP]
Out: { {0,1,0}, {1,0,0}, {0,1,1}, {1,1,0}, {1,1,1}, {1,0,1}, {0,0,1} }
In: c = {{1, 0, 0}, {0, 0, 0}, {1, 0, 0}, {0, 1, 1}, {1, 0, 1}, {0, 0, 0}, {0, 0, 0}};
In: BCHRSCalcSyndromes[ALT, c, 4]
Out: {{0, 0, 0}, {1, 0, 1}, {1, 0, 0}, {0, 1, 0}}

5.4.25 Výpočet Inverzní matice k matici s prvky GF

Funkce GFInverseMatrix přijme matici, jejíž prvky jsou vektory, představující prvky GF. Tuto matici převede do formátu GF implementovaného v jazyce Wolfram. Tam se pomocí funkce Inverse spočte inverze. Poté se matice převede zpět. K převodu tam se použije pomocná funkce vect2GF a k převodu zpět funkce GF2vect.

Funkce vyžaduje globální proměnnou IP, ve které je ireducibilní polynom konečného tělesa GF! Funkce používá package FiniteFields!

Příklad 40:

In: alpha = {0,1,0};
In: IP = {1,0,1,1};
In: ALT = AntiLogTable[alpha,IP]
Out: { {0,1,0}, {1,0,0}, {0,1,1}, {1,1,0}, {1,1,1}, {1,0,1}, {0,0,1} }
In: matrix = {{{1, 1, 1}, {1, 0, 0}}, {{0, 1, 1}, {1, 1, 0}}};
In: matrix // MatrixForm
Out:

$$\begin{pmatrix} \{1,1,1\} & \{1,0,0\} \\ \{0,1,1\} & \{1,1,0\} \end{pmatrix}$$

In: RSCalcSyndromes[ALT, c, 4] // MatrixForm

$$\begin{pmatrix} \{0,1,0\} & \{1,0,1\} \\ \{0,0,1\} & \{1,0,0\} \end{pmatrix}$$

5.4.26 Násobení matice s vektorem s prvky z GF

Funkce GFMatrixMul přijme matici a vektor a ty vynásobí dle pravidel násobení matic, ale místo klasických operací sčítání a násobení se použijí ty pro práci s GF.

Příklad 41:

In: alpha = {0,1,0};
In: IP = {1,0,1,1};
In: ALT = AntiLogTable[alpha,IP]
Out: { {0,1,0}, {1,0,0}, {0,1,1}, {1,1,0}, {1,1,1}, {1,0,1}, {0,0,1} }
In: matrix = {{{0, 1, 0}, {1, 0, 1}}, {{0, 0, 1}, {1, 0 , 0}}};
In: matrix // MatrixForm
Out:

$$\begin{pmatrix} \{0,1,0\} & \{1,0,1\} \\ \{0,0,1\} & \{1,0,0\} \end{pmatrix}$$

In: vector = {{0, 0, 0}, {1, 0, 1}};
In: GFMatrixMul[ALT, matrix, vector]
Out: {{1, 1, 1}, {0, 1, 0}}

5.4.27 Výpočet lokalizačního polynomu pro RS/BCH

Funkce FindLocator[ALT,syn] přijme tabulku antilogaritmů a seznam syndromů. Sestrojí matici $Theta$, vypočte její inverzi pomocí funkce GFInverseMatrix. Maticovým násobením pro prvky GF (funkce GFMatrixMul) vynásobí matici $Theta^{-1}$ vektorem syndromů $s_\nu + 1, ...s2\nu$. Za tento vektor ještě připojí jedničkový prvek (poslední v tabulce antilogaritmů).

Příklad 42:

In: alpha = {0,1,0};
In: IP = {1,0,1,1};
In: ALT = AntiLogTable[alpha,IP];
In: syndromes = {{0, 0, 0}, {1, 0, 1}, {1, 0, 0}, {0, 1, 0}};
In: FindLocator[ALT, syndromes]
Out: {{0, 0, 1}, {0, 1, 1}, {0, 0, 1}}

5.4.28 Nalezení pozic chyb z lokalizačního polynomu

Funkce přijme tabulku antilogaritmů, lokalizační polynom a délku přijaté zprávy. Chienovým vyhledáváním se naleznou kořeny lokalizačního polynomu. Pomocí tabulky antilogaritmů se spočítají logaritmy kořenů. Ty udávají pozice chyb. Vloží se do seznamu a ten je pak výstupem této funkce. Čísla ve výstupním seznamu udávají mocniny x, ve kterých je chyba. Jinými slovy jsou to pozice znaků zprava, počítané od nuly.

Příklad 43:

In: alpha = {0,1,0};
In: IP = {1,0,1,1};
In: ALT = AntiLogTable[alpha,IP];
In: locator = {{0, 0, 1}, {0, 1, 1}, {0, 0, 1}};
In: RSFindErrors[ALT, locator,7]
Out: {5,2}

5.4.29 Výpočet velikosti chyby pro RS

Vstupem funkce RSFindMagnitude je tabulka antilogaritmů, seznam pozic chyb a seznam syndromů. Funkce sestrojí matici Ω, poté její inverzi pomocí funkce Inverse. Pomocí násobení matic s prvky GF provede násobení vektorem příslušných syndromů (syndromy s_1 až s_ν). Výstupem je seznam velikostí jednotlivých chyb v pořadí odpovídajícím pozicím ze vstupu.

Příklad 44:

Tabulka 5.1: Tabulka možných generujících prvků pro tvorbu EG-LDPC [5]

s	n	k	min kvzd	IP pro GF
2	15	7	5	$x^4 + x + 1$
3	63	37	9	$x^6 + x + 1$
4	255	175	17	$x^8 + x^4 + x^3 + x^2 + 1$
5	1023	781	33	$x^{10} + x^3 + 1$
6	4095	3367	65	$x^{12} + x^6 + x^4 + x + 1$
7	16383	14197	129	$x^{14} + x^5 + x^3 + x + 1$

In: alpha = {0,1,0};
In: IP = {1,0,1,1};
In: ALT = AntiLogTable[alpha,IP];
In: errorPositions = {5,2}
In: syndromes = {{0, 0, 0}, {1, 0, 1}, {1, 0, 0}, {0, 1, 0}};
In: RSFindMagnitude[ALT, errorPositions,syndromes]
Out: {{1,1,1},{0,1,0}}

5.4.30 Oprava přijaté zprávy

Funkce dostane na vstup přijatou zprávu, seznam pozic chyb a seznam velikostí chyb. Funkce projde seznam pozic chyb a na každé pozici chyby přičte ke zprávě c příslušnou velikost chyby. Tím je slovo opraveno.

Příklad 45:

In: c = {{1, 0, 0}, {0, 0, 0}, {1, 0, 0}, {0, 1, 1}, {1, 0, 1}, {0, 0, 0}, {0, 0, 0}};
In: errorPositions = {5,2};
In: magnitude={{1,1,1},{0,1,0}};
In: RSCorrect[c,errorPositions,magnitude]
Out: {{1, 0, 0}, {1, 1, 1}, {1, 0, 0}, {0, 1, 1}, {1, 1, 1}, {0, 0, 0}, {0, 0, 0}};

5.5 LDPC

5.5.1 Tvorba prvního řádku matice

Vstupem funkce je tabulka antilogarimů. Ireducibilní polynom této tabulky musí být stupně $2s$ a generující prvek musí mít řád $2^{2s} - 1$. Některé vhodné generující prvky těles pro tvorbu kódu shrnuje tabulka 5.1

Příklad 46:

In: alpha = {0,0,1,0};
In: IP = {1,0,0,1,1};
In: ALT=AntiLogTable[alpha,IP];
In: LDPCGEN[ALT]
Out: {1,0,0,0,1,0,1,1,0,0,0,0,0,0,0}

5.5.2 Bitflipping algoritmus

Funkce LDPCdecode dostane na vstup kontrolní matici, přijaté slovo c a maximální počet iterací. Přijaté slovo se upravuje pomocí bitflipping algoritmu dokud se mění, nebo dokud nedoběhne počet iterací. Funkce vypisuje, který uzel kam posílá jaká data.

Příklad 47:

In: H= {{1, 1, 0, 1, 0, 0}, {0, 1, 1, 0, 1, 0}, {1, 0, 0, 0, 1, 1}, {0, 0, 1, 1, 0, 1}};
In: H // MatrixForm
Out:

$$H = \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix} \tag{5.1}$$

In: maxiterations = 10;
In: c= {0, 1, 1, 0, 0, 1};
In: LDPCdecode[H, c, maxiterations]
Out:

STEP: 1. codeword = {0,1,1,0,0,1}

CNode 1 Received {0,1,0} from {1,2,4}nodes and replies {1,0,1}

CNode 2 Received {1,1,0} from {2,3,5}nodes and replies {1,1,0}

CNode 3 Received {0,0,1} from {1,5,6}nodes and replies {1,1,0}

CNode 4 Received {1,0,1} from {3,4,6}nodes and replies {1,0,1}

VNode 1 with original value 0 received {1,1}. Majority is 1

VNode 2 with original value 1 received {0,1}. Majority is 1

VNode 3 with original value 1 received {1,1}. Majority is 1

VNode 4 with original value 0 received {1,0}. Majority is 0

VNode 5 with original value 0 received {0,1}. Majority is 0

VNode 6 with original value 1 received {0,1}. Majority is 1

{0,1,1,0,0,1} Codeword before iteration

{1,1,1,0,0,1} Codeword after iteration

Codeword changed. Repeat.

————End of step 1 ————

STEP: 2. codeword = {1,1,1,0,0,1}

CNode 1 Received {1,1,0} from {1,2,4}nodes and replies {1,1,0}

CNode 2 Received {1,1,0} from {2,3,5}nodes and replies {1,1,0}

CNode 3 Received {1,0,1} from {1,5,6}nodes and replies {1,0,1}

CNode 4 Received {1,0,1} from {3,4,6}nodes and replies {1,0,1}

VNode 1 with original value 1 received {1,1}. Majority is 1

VNode 2 with original value 1 received {1,1}. Majority is 1

VNode 3 with original value 1 received {1,1}. Majority is 1

VNode 4 with original value 0 received {0,0}. Majority is 0

VNode 5 with original value 0 received {0,0}. Majority is 0

VNode 6 with original value 1 received {1,1}. Majority is 1

{1,1,1,0,0,1} Codeword before iteration

{1,1,1,0,0,1} Codeword after iteration

Codeword is same. Terminate.

————End of step 2 ————

Terminated: codeword doesnt change.

5.6 Prokládaný kód

5.6.1 Tvorba generovacího / kontrolního mnohočlenu

Funkce přijme generující/kontrolní polynom prokládaného kódu a délku shluku, proti kterému se chceme bránit. Mezi každé dva koeficienty původního generátoru je vloženo $m - 1$ nulových koeficientů pomocí funkce Insert.

Příklad 48:

In: Gx = {1,0,1,1};
In: m=3;
In: Interleaved[Gx,m]
Out: {1,0,0,0,0,0,1,0,0,1}

Příklad 49:

In: Hx = {1,0,1,1,1};
In: m=3;
In: Interleaved[Hx,m]
Out: {1,0,0,0,0,0,1,0,0,1,0,0,1}

Kapitola 6

Testování

Správnost implementace kódů jsem se rozhodl ověřit spočítáním stejných příkladů v MATLABu a ve Wolfram Mathematice. Případně kódy, které nejsou podporované, alespoň ověřit, zda opravují předpokládaný počet chyb. Pro testování byly mimo jiné použity příklady ze slidů pro předmět MI-AAK dostupné na https://edux.fit.cvut.cz/courses/MI-AAK.

6.1 Testování vybraných dílčích funkcí

6.1.1 Tvorba generující matice z paritní

Příklad 50:

In: G = {{1, 0, 0, 0, 1, 0, 1}, {{0, 1, 0, 0, 1, 1, 1}, {{0, 0, 1, 0, 1, 1, 0}, {{0, 0, 0, 1, 1, 0, 1}};
In: G // MatrixForm
Out:

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

In: gen2par[G]//MatrixForm
Out:

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

6.1.2 Tvorba paritní matice z generující

Příklad 51:

In: H = {{1, 1, 1, 0, 1, 0, 0}, {{0, 1, 1, 1, 0, 1, 0}, {{0, 0, 1, 0, 1, 1, 0}, {{1, 1, 0, 1, 0, 0, 1}};

In: H // MatrixForm
Out:

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

In: par2gen[H]//MatrixForm
Out:

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

6.2 Hammingův kód

V této práci je Hammingův kód tvořen jinak než v MATLABu. Tam je tvořen jako cyklický kód. Nicméně v případě jedné chyby nebo přenosu bez chyb se kód chová stejně jako kód popisovaný v MATLABu. Chování v případě více chyb není důležité, protože proti více chybám kód nezabezpečuje a stejně zprávu opraví na špatné kódové slovo.

Příklad 52: Zadání:
Binární Hammingův kód (opravující 1 chybu). Parametr r = 3 (a tedy k=4, n=7). Zpráva a = 1001. Během přenosu dojde k chybě e = 0010000.

Řešení Wolfram:
In: h = HammGen[3];
h // MatrixForm Out:

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

g = par2gen[h];
g // MatrixForm
Out:

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

In: a = {1, 0, 0, 1};
In: b = Code[a, g]
Out: {1, 0, 0, 1, 1, 0, 0}

In: e = {0, 0, 1, 0, 0, 0, 0};
In: c = b+e;
Out: {1, 0, 1, 1, 1, 0, 0}
In: Correct[c,h]
Out: {1, 0, 0, 1, 1, 0, 0}
In: Unencode[%,3]
Out: {1, 0, 0, 1}

Opravené kódové slovo oříznuté o kontrolní bity se shoduje se zprávou a. Funkce zakódovala zprávu tak, že byla schopna opravit jednu chybu. Testování dále probíhalo pro případ, v nemž je zpráva bez chyb. Všechna testování byla úspěšná.

6.3 Binární BCH kód

Příklad 53:

Zadání:
Chceme kód pro opravu 2 chyb (kvzd ≥ 5, $\delta = 5$). Ireducibilní polynom $= x^4 + x + 1$. Odesílané slovo $a = 1000010$. Během přenosu dojde k chybě $e = 011000000000000$. Úkolem je spočítat syndromy, najít lokalizační polynom, určit pozice chyb a chyby opravit.

Řešení:
In: IP = {1, 0, 0, 1, 1};
In: ALT = AntiLogTable[{0, 0, 1, 0}, IP];
In: g = BCHGeneratorPoly[ALT, 5]
Out: {1, 1, 1, 0, 1, 0, 0, 0, 1}
In: a = {1, 0, 0, 0, 0, 1, 0};
In: b = SysCode[a, g]
Out: {1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1}
In: e = {0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
In: c = b + e
Out: {1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1}
In: listc = Array[c[[#1]] &, {15, 1}](*Obalí prvky c do malých seznamů*)
Out: {{1}, {1}, {1}, {0}, {0}, {1}, {0}, {1}, {0}, {0}, {1}, {1}, {0}, {1}, {1}}
In: syn = BCHRSCalcSyndromes[ALT, listc, 4]
Out: {{0, 0, 1, 0}, {0, 1, 0, 0}, {0, 1, 1, 0}, {0, 0, 1, 1}}
In: FindLocator[ALT, syn]
Out: {{0, 1, 1, 1}, {0, 0, 1, 0}, {0, 0, 0, 1}}
In: RSFindErrors[ALT, %, Length[c]]
Out: {13, 12}

In: BCHCorrect[c, %]
Out: {1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1}
In: Unencode[%, Length[g] - 1]
Out: = {1, 0, 0, 0, 0, 1, 0}
In: % == a
Out:= True

6.4 RS kód

Příklad 54:

Zadání:
RS-kód nad tělesem $GF(2^3)$ s IP $= X^3 + x + 1$ opravující 2 chyby.
Tedy $\delta = 5$, $r = 4$.
Zpráva a = 474. Během přenosu dojde k chybě e=0702000.

Řešení:
In: IP = {1, 0, 1, 1};
In: ALT = AntiLogTable[0, 1, 0, IP];
In: n = 7;
In: r = 4;
In: G = GFromALT[ALT, r, n] // MatrixForm
Out:

$$\begin{pmatrix} 1 & 0 & 0 & 6 & 1 & 6 & 7 \\ 0 & 1 & 0 & 4 & 1 & 5 & 5 \\ 0 & 0 & 1 & 3 & 1 & 2 & 3 \end{pmatrix}$$

In: H = HFromALT[ALT, r, n] // MatrixForm
Out:

$$\begin{pmatrix} 5 & 7 & 6 & 3 & 4 & 2 & 1 \\ 7 & 3 & 2 & 5 & 6 & 4 & 1 \\ 6 & 2 & 7 & 4 & 5 & 3 & 1 \\ 3 & 5 & 4 & 7 & 2 & 6 & 1 \end{pmatrix}$$

In: a = {"4", "7", "4"}
Out: {4, 7, 4}
In: a = UnhexForm[a, Length[IP] - 1];
Out: {{1, 0, 0}, {1, 1, 1}, {1, 0, 0}}
In: b = RSEncodeMsg[ALT, a, r];
In: b // HexForm
Out: {4, 7, 4, 3, 7, 0, 0}
In: e = {{0, 0, 0}, {1, 1, 1}, {0, 0, 0}, {0, 1, 0}, {0, 0, 0}, {0, 0, 0}, {0, 0, 0}};
In: e // HexForm
Out: {0, 7, 0, 2, 0, 0, 0}
In: c = GFPolyAdd[b, e];

In: c // HexForm
Out: {4, 0, 4, 1, 7, 0, 0}
In: syn = RSCalcSyndromes[ALT, c, r];
In: syn // HexForm
Out: {5, 3, 6, 3}
In: locator = FindLocator[ALT, syn];
In: locator // HexForm
Out: {2, 4, 1}
In: locator // HexForm // HexPolForm
Out: $2x^2 + 4x + 1$ In: errors = RSFindErrors[ALT, locator, Length[c]](*pozice chyb udana mocninami - tj pocitano zprava od nuly*)
Out: {5,3}
In: magnitude = RSFindMagnitude[ALT, errors, syn];
In: magnitude // HexForm
Out: {7,2}
In: RSCorrect[c, errors, magnitude] // HexForm
Out: {4, 7, 4, 3, 7, 0, 0}
In: Unencode[%, r]
Out: Out: {4, 7, 4}
In: % == (a // HexForm)
Out: True

Kód správně opravil obě chyby. Další testování probíhalo s 10 chybami, s méně chybami než je maximum opravitelných, včetně přenosu bez chyby. Ve všech případech testování potvrdilo správnost implementace.

6.5 LDPC kód

6.5.1 Tvorba paritní matice LDPC kódu pomocí EG

Tvorbu LDPC matic pomocí Eukleidovské geometrie MATLAB nepodporuje. Nicméně ze vzniklé matice je zřejmé, že odpovídá počet 1 ve sloupcích i v řádcích a že nikde nejsou cykly délky 4.

Příklad 55:

In: alpha = {0,0,1,0};
In: IP = {1,0,0,1,1};
In: ALT=AntiLogTable[alpha,IP];
In: LDPCGEN[ALT]
Out: {1,0,0,0,1,0,1,1,0,0,0,0,0,0,0}
In: MatFromVect[%]

Out:

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

Testování bylo ještě provedeno pro s=3. Větší matice už byly příliš nepřehledné.

6.5.2 Testování kódování a dekódování za použití bitflipping algoritmu

Příklad 56:

In: IP = 1, 0, 0, 1, 1;
In: ALT = AntiLogTable[0, 0, 1, 0, 1, 0, 0, 1, 1];
In: H = LDPCH[LDPCGEN[ALT]];
In: G = par2gen[H];
In: a = 1, 0, 1, 1, 1, 1, 1;
In: b = SysCode[a, g]
Out: {1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1}
In: e = {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0};
In: c = PolynomialMod[b + e, 2]
Out: {0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1}
In: LDPCdecode[H, c, 10]
Out:
STEP: 1. codeword = {0,0,1,1,1,1,1,1,0,0,0,1,1,1,1}

CNode 1 Received {0,1,1,1} from {1,5,7,8}nodes and replies {1,0,0,0}

CNode 2 Received {1,1,1,1} from {4,6,7,15}nodes and replies {1,1,1,1}

CNode 3 Received {1,1,1,1} from {3,5,6,14}nodes and replies {1,1,1,1}

CNode 4 Received {0,1,1,1} from {2,4,5,13}nodes and replies {1,0,0,0}

CNode 5 Received {0,1,1,1} from {1,3,4,12}nodes and replies {1,0,0,0}

CNode 6 Received {0,1,0,1} from {2,3,11,15}nodes and replies {0,1,0,1}

CNode 7 Received {0,0,0,1} from {1,2,10,14}nodes and replies {1,1,1,0}

CNode 8 Received {0,0,1,1} from {1,9,13,15}nodes and replies {0,0,1,1}

CNode 9 Received {1,1,1,1} from {8,12,14,15}nodes and replies {1,1,1,1}

CNode 10 Received {1,0,1,1} from {7,11,13,14}nodes and replies {0,1,0,0}

CNode 11 Received {1,0,1,1} from {6,10,12,13}nodes and replies {0,1,0,0}

CNode 12 Received {1,0,0,1} from {5,9,11,12}nodes and replies {1,0,0,1}

CNode 13 Received {1,1,0,0} from {4,8,10,11}nodes and replies {1,1,0,0}

CNode 14 Received {1,1,0,0} from {3,7,9,10}nodes and replies {1,1,0,0}

CNode 15 Received {0,1,1,0} from {2,6,8,9}nodes and replies {0,1,1,0}

VNode 1 with original value 0 received {1,1,1,0}. Majority is 1

VNode 2 with original value 0 received {1,0,1,0}. Majority is 0

VNode 3 with original value 1 received {1,0,1,1}. Majority is 1

VNode 4 with original value 1 received {1,0,0,1}. Majority is 1

VNode 5 with original value 1 received {0,1,0,1}. Majority is 1

VNode 6 with original value 1 received {1,1,0,1}. Majority is 1

VNode 7 with original value 1 received {0,1,0,1}. Majority is 1

VNode 8 with original value 1 received {0,1,1,1}. Majority is 1

VNode 9 with original value 0 received {0,0,0,0}. Majority is 0

VNode 10 with original value 0 received {1,1,0,0}. Majority is 0

VNode 11 with original value 0 received {0,1,0,0}. Majority is 0

VNode 12 with original value 1 received {0,1,0,1}. Majority is 1

VNode 13 with original value 1 received {0,1,0,0}. Majority is 0

VNode 14 with original value 1 received {1,0,1,0}. Majority is 1

VNode 15 with original value 1 received {1,1,1,1}. Majority is 1

{0,0,1,1,1,1,1,1,0,0,0,1,1,1,1} Codeword before iteration

{1,0,1,1,1,1,1,1,0,0,0,1,0,1,1} Codeword after iteration

Codeword changed. Repeat.

————End of step 1 ————

STEP: 2. codeword = {1,0,1,1,1,1,1,1,0,0,0,1,0,1,1}

CNode 1 Received {1,1,1,1} from {1,5,7,8}nodes and replies {1,1,1,1}

CNode 2 Received {1,1,1,1} from {4,6,7,15}nodes and replies {1,1,1,1}

CNode 3 Received {1,1,1,1} from {3,5,6,14}nodes and replies {1,1,1,1}

CNode 4 Received {0,1,1,0} from {2,4,5,13}nodes and replies {0,1,1,0}

CNode 5 Received {1,1,1,1} from {1,3,4,12}nodes and replies {1,1,1,1}

CNode 6 Received {0,1,0,1} from {2,3,11,15}nodes and replies {0,1,0,1}

CNode 7 Received {1,0,0,1} from {1,2,10,14}nodes and replies {1,0,0,1}

CNode 8 Received {1,0,0,1} from {1,9,13,15}nodes and replies {1,0,0,1}

CNode 9 Received {1,1,1,1} from {8,12,14,15}nodes and replies {1,1,1,1}

CNode 10 Received {1,0,0,1} from {7,11,13,14}nodes and replies {1,0,0,1}

CNode 11 Received {1,0,1,0} from {6,10,12,13}nodes and replies {1,0,1,0}

CNode 12 Received {1,0,0,1} from {5,9,11,12}nodes and replies {1,0,0,1}

CNode 13 Received {1,1,0,0} from {4,8,10,11}nodes and replies {1,1,0,0}

CNode 14 Received {1,1,0,0} from {3,7,9,10}nodes and replies {1,1,0,0}

CNode 15 Received {0,1,1,0} from {2,6,8,9}nodes and replies {0,1,1,0}

VNode 1 with original value 1 received {1,1,1,1}. Majority is 1

VNode 2 with original value 0 received {0,0,0,0}. Majority is 0

VNode 3 with original value 1 received {1,1,1,1}. Majority is 1

VNode 4 with original value 1 received {1,1,1,1}. Majority is 1

VNode 5 with original value 1 received {1,1,1,1}. Majority is 1

VNode 6 with original value 1 received {1,1,1,1}. Majority is 1

VNode 7 with original value 1 received {1,1,1,1}. Majority is 1

VNode 8 with original value 1 received {1,1,1,1}. Majority is 1

VNode 9 with original value 0 received {0,0,0,0}. Majority is 0

VNode 10 with original value 0 received {0,0,0,0}. Majority is 0

VNode 11 with original value 0 received {0,0,0,0}. Majority is 0

VNode 12 with original value 1 received {1,1,1,1}. Majority is 1

VNode 13 with original value 0 received {0,0,0,0}. Majority is 0

VNode 14 with original value 1 received {1,1,1,1}. Majority is 1

VNode 15 with original value 1 received {1,1,1,1}. Majority is 1

{1,0,1,1,1,1,1,1,0,0,0,1,0,1,1} Codeword before iteration

{1,0,1,1,1,1,1,1,0,0,0,1,0,1,1} Codeword after iteration

Codeword is same. Terminate.

————End of step 2 ————

Terminated: codeword doesnt change.

Kód odhalil obě chyby.

6.5.3 Test dekódování za použití bitflipping algoritmu na matici s cyklem délky 4

Tento test má názorně ukázat, co se může stát, pokud budou v paritní matici cykly délky 4.

Příklad 57:

In: H= {{1, 1, 0, 1, 0, 0}, {0, 1, 1, 0, 1, 0}, {1, 0, 0, 0, 1, 1}, {0, 0, 1, 1, 0, 1}};
In: H // MatrixForm
Out:

$$H = \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 \end{pmatrix} \tag{6.1}$$

In: maxiterations = 2;
In: c= {0, 1, 1, 0, 0, 1};
In: LDPCdecode[H, c, maxiterations]
Out:

STEP: 1. codeword = {0,1,1,0,0,1}

CNode 1 Received {0,1,0} from {1,2,4}nodes and replies {1,0,1}

CNode 2 Received {0,1,0} from {1,2,5}nodes and replies {1,0,1}

CNode 3 Received {1,0,1} from {3,5,6}nodes and replies {1,0,1}

CNode 4 Received {1,0,1} from {3,4,6}nodes and replies {1,0,1}

VNode 1 with original value 0 received {1,1}. Majority is 1

VNode 2 with original value 1 received {0,0}. Majority is 0

VNode 3 with original value 1 received {1,1}. Majority is 1

VNode 4 with original value 0 received {1,0}. Majority is 0

VNode 5 with original value 0 received {1,0}. Majority is 0

VNode 6 with original value 1 received {1,1}. Majority is 1

{0,1,1,0,0,1} Codeword before iteration

{1,0,1,0,0,1} Codeword after iteration

Codeword changed. Repeat.

————End of step 1 ————

STEP: 2. codeword = {1,0,1,0,0,1}

CNode 1 Received {1,0,0} from {1,2,4}nodes and replies {0,1,1}

CNode 2 Received {1,0,0} from {1,2,5}nodes and replies {0,1,1}

CNode 3 Received {1,0,1} from {3,5,6}nodes and replies {1,0,1}

CNode 4 Received {1,0,1} from {3,4,6}nodes and replies {1,0,1}

VNode 1 with original value 1 received {0,0}. Majority is 0

VNode 2 with original value 0 received {1,1}. Majority is 1

VNode 3 with original value 1 received {1,1}. Majority is 1

VNode 4 with original value 0 received {1,0}. Majority is 0

VNode 5 with original value 0 received {1,0}. Majority is 0

VNode 6 with original value 1 received {1,1}. Majority is 1

{1,0,1,0,0,1} Codeword before iteration

{0,1,1,0,0,1} Codeword after iteration

Codeword changed. Repeat.

————End of step 2 ————

Terminated: Max steps reached.

V prvním kroku se slovo změní z {0,1,1,0,0,1} na {1,0,1,0,0,1}. V druhém se změní zpátky z {1,0,1,0,0,1} na {0,1,1,0,0,1}. Pokud by algoritmus běžel dál, budou se opakovat pouze tyto dva kroky. S touto maticí tedy není bitflipping algoritmus schopný chybu opravit.

Závěr

První část mé diplomové práce seznamuje s teorií a možnostmi bezpečnostních kódů. Pro dekódování RS a binárních BCH kódů je vysvětlen a použit algoritmus Peterson-Gorenstein-Zierler. Pro prezentaci LDPC kódů byla vybrána metoda konstrukce paritních matic pomocí Eukleidovské geometrie. Jako dekódovací metoda byl prezentován bitflipping algoritmus(hard-decision přístup). Během testování bylo poukázáno na to, co mohou udělat cykly délky 4 v LDPC kódech. Výstupem praktické části je knihovna funkcí pro program Wolfram Mathematica a soubor příkladů ze školních slidů řešených v této knihovně. Tato část může sloužit jako ukázka fungování bezpečnostních kódů. Knihovna byla otestována za použití příkladů ze cvičení a funguje.

Literatura

[1] Pluháček Alois, Materiály k předmětu AAK na ČVUT v Praze školní rok 2014/2015 , cit 2016-11-04. Dostupné z WWW <https://edux.fit.cvut.cz/courses/MI-AAK/_media/lectures/04k4_1-mi-aak.pdf>

[2] Jančařík Antonín, Algebra v informatice [online], cit 2016-03-05. Dostupné z WWW <http://class.pedf.cuni.cz/jancarik/download/AvI.pdf>

[3] Olšák Petr, Materiály k předmětu Lineární Algebra [online], cit 2016-10-02. Dostupné z WWW <http://petr.olsak.net/bilin/kody4.pdf>

[4] Pluháček Alois, Materiály k předmětu BJK rok 2011, cit 2016-11-04.

[5] Watson E.J. Primitive polynomial mod 2 [online], cit 2016-23-04. Dostupné z WWW <http://www.ams.org/journals/mcom/1962-16-079/S0025-5718-1962-0148256-1/S0025-5718-1962-0148256-1.pdf>

[6] Johnson Sarah, Introducing Low-Density Parity-Check Codes [online], cit 2016-14-03. Dostupné z WWW <http://sigpromu.org/sarah/SJohnsonLDPCintro.pdf>

[7] Yu Kou and Shu Lin, Low Density Parity Check Codes Based on Finite Geometries: A Rediscovery and New Results [online], cit 2016-14-03. Dostupné z WWW <http://web.stanford.edu/group/cioffi/ee379b/class_reader/ucd1.pdf>

[8] Hrouza Ondřej, LDPC kódy: diplomová práce. Brno: Vysoké učení technické v Brně [online], cit 2016-20-02. Dostupné z WWW <https://www.vutbr.cz/www_base/zav_prace_soubor_verejne.php?file_id=52086>

[9] Kaiser Tomáš, Učební text k předmětu samoopravné kódy na ZČU v Plzni v ZS 2011/12 [online], cit 2016-02-05. Dostupné z WWW <http://home.zcu.cz/ kaisert/kody/kody.pdf>

Příloha A

Seznam použitých zkratek

ECC Error Correcting Code

FEC Forward Error Correcting

LDPC Low Density Parity Check

RS kód Reed-Solomon kód

SEC Single Error Correcting

kvzd kódová vzdálenost

DVB Digital Video Broadcasting

ATSC Advanced Television Systems Committee

DSL Digital Subscriber Line

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `kody.nb` ..... notebook s funkcemi
- `priklady.nb` ..... notebook s příklady
- `src`
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `DP_Doubek_Jakub.pdf` ..... text práce ve formátu PDF