FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Pokročilé bezpečnostní kódy v programu Wolfram Mathematica
Student:	Bc. Stanislav Koleník
Vedoucí:	Ing. Pavel Kubalík, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Počítačová bezpečnost
Katedra:	Katedra informační bezpečnosti
Platnost zadání:	do konce letního semestru 2021/2022

Pokyny pro vypracování

1) Rozšiřte již existující sadu výukových scénářů pro podporu výuky bezpečnostních kódů o vybrané pokročilé kódy.
2) Dodržte členění na výukové scénáře a balíčky a zachovejte jejich formát.
3) Zaměřte se především na Fireovy kódy, součinové kódy, RM kódy, nebinární BCH kódy a RS kódy, Goppa kódy a kódy konvoluční.
4) V dostatečné míře nastudujte a zdokumentujte příslušnou matematickou teorii.
5) Navržené řešení zrealizujte a řádně otestujte.
6) Vytvořte několik příkladů použití každého kódu.
7) Využijte získaných poznatků ke zmapování současného stavu použití bezpečnostních kódů v kryptografii.

Elektronicky schválil/a prof. Ing. Róbert Lórencz, CSc. dne 4. února 2021 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Pokročilé bezpečnostní kódy v programu Wolfram Mathematica

Bc. Stanislav Koleník

Katedra informační bezpečnosti
Vedoucí práce: Ing. Pavel Kubalík, Ph.D.

6. května 2021

Poděkování

Děkuji Ing. Pavlu Kubalíkovi, Ph.D. za opětovný svědomitý a aktivní přístup k vedení mé práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl 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, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 6. května 2021 ……

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2021 Stanislav Koleník. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci

Koleník, Stanislav. *Pokročilé bezpečnostní kódy v programu Wolfram Mathematica.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

Abstrakt

Bezpečnostní kódy, také označované jako samoopravné, se používají v digitálních komunikačních systémech k zabezpečení dat před šumem v průběhu přenosu. Existuje mnoho metod, jak dosáhnout této ochrany, všechny jsou matematické povahy. V nedávné minulosti byla vytvořena sada výukových materiálů v systému Wolfram Mathematica demonstrující některé z těchto metod. Tato práce sadu rozšiřuje o vybrané pokročilé kódy.

Klíčová slova Fireovy kódy, RM kódy, Goppa kódy, BCH kódy, RS kódy, součinové kódy, konvoluční kódy, podpora výuky, Wolfram Mathematica

Abstract

Error-control codes are used in digital communication systems to protect data against noise during transmission. There are many methods to achieve this kind of protection, all are mathematical in nature. A set of teaching materials in the Wolfram Mathematica computing system has been developed in the past to demonstrate some of these methods. The aim of this work is to extend the set by adding some more advanced codes.

Keywords Fire codes, RM codes, Goppa codes, BCH codes, RS codes, product codes, convolutional codes, teaching resources, Wolfram Mathematica

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Bezpečnostní neboli samoopravné kódy umožňují ochránit digitální data před šumem v průběhu jejich přenosu či uložení. Vlivem šumu totiž může dojít k invertování některých z přenášených či uložených bitů. V případě přenosu je zpráva před odesláním modifikována bezpečnostním kódem tak, že do ní jsou určitým způsobem vloženy redundantní bity. Tím sice dojde k navýšení délky zprávy a je nutné přenášet větší objem dat, příjemce zprávy však díky této úpravě může detekovat, zdali byla přijatá zpráva zasažena chybou, či nikoliv. Redundantní bity mu dokonce mohou poskytnout dostatek informací k tomu, aby byl schopen určitý počet chyb opravit, a zrekonstruovat tak původní zprávu. Při ukládání dat je postup analogický.

Na Fakultě informačních technologií ČVUT v Praze je těmto kódům věnován předmět *Bezpečnostní kódy*. Především pro podporu výuky tohoto předmětu byla v nedávné minulosti vyvinuta sada digitálních výukových materiálů automatizujících vybrané výpočetní operace, aby je nebylo nutné provádět zdlouhavě ručně a bylo tak možné v omezeném čase stihnout více příkladů. Zároveň může pomůcka sloužit vyučujícímu ke generování nových testových otázek a jejich následné kontrole, případně jako podklad pro další závěrečné práce.

Cílem předkládané práce je rozšířit tuto sadu výukových materiálů o další pokročilejší kódy a zachovat přitom důraz na jejich srozumitelnost a názornost. Předpokládané využití nově zpracovaných materiálů by mělo být stejné jako u těch stávajících. Navíc mají být získané zkušenosti s prací s kódy využity pro shrnutí aktuálního stavu aplikace bezpečnostních kódů v kryptografii. Za aktuální je považována doba vzniku této práce, tedy začátek roku 2021.

Bližší popis existujících studijních materiálů, konkrétní zpracované kódy a kódy, o než mají být materiály rozšířeny, jsou uvedeny v kapitole 1. Následně je v kapitole 2 navrženo, jakým způsobem bude rozšíření provedeno. Součástí návrhu je členění materiálů dle jednotlivých kódů. Většina kódů však používá podobné elementární operace, které budou vyčleněny a sdíleny všemi

kódy v podobě tzv. pomocných balíčků. Realizaci těchto sdílených operací je věnována kapitola 3. Hlavní náplní tohoto textu je kapitola 4 rozebírající jednotlivé bezpečnostní kódy a jejich implementaci. Na závěr je v kapitole 5 shrnut aktuální vývoj standardizace post-kvantových kryptosystémů a role bezpečnostních kódů v rámci něj.

Tato struktura je lehce nestandardní. Běžně by analýza a teoretický popis jednotlivých kódů předcházely návrhu. A ve skutečnosti mu také předcházely. Ačkoliv jsou však všechny kódy založené na podobných principech, detaily jejich fungování se mohou výrazně lišit a každý z pokročilých kódů tvoří víceméně samostatné téma. Proto je text organizován spíše než podle pracovního postupu právě podle jednotlivých kódů. Veškeré informace o libovolném z kódů jsou uspořádány do jediné podkapitoly v kapitole číslo 4. Případně mohou odkazovat do kapitoly číslo 3 na některou ze společných vlastností.

Kapitola 1

Vymezení cíle a návaznost na předchozí práce

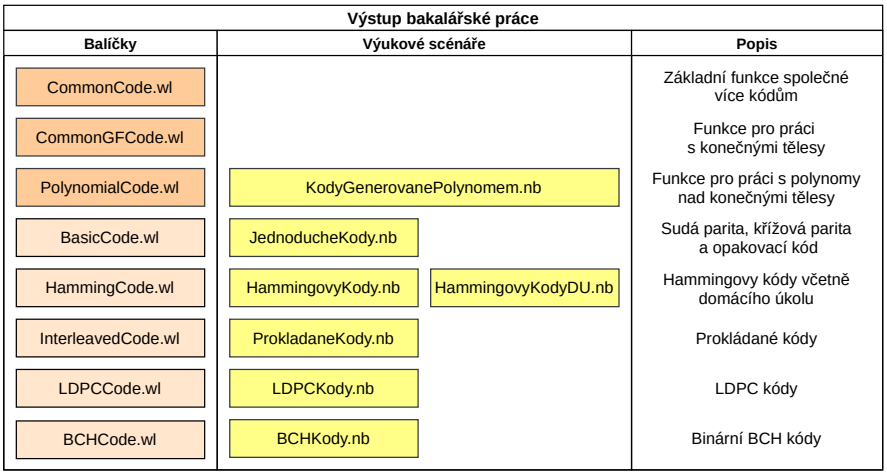
Cílem práce je rozšířit existující sadu výukových pomůcek napsaných v systém Wolfram Mathematica pro demonstraci fungování bezpečnostních neboli samoopravných kódů. Sada byla vytvořena v rámci bakalářské práce pod názvem *Podpora výuky bezpečnostních kódů v programu Wolfram Mathematica* [1], ta bude dále označována zkráceně jako BP. Motivací k jejímu vzniku byla potřeba automatizovat vybrané výpočty, aby je v rámci výuky nebylo nutné počítat zdlouhavě ručně, a mohlo tak být ukázáno více příkladů, které by studenta blíže seznámily s fungováním daného kódu. Současně nebyly nalezeny žádné uspokojivé implementace bezpečnostních kódů v preferovaném výpočetním systému Mathematica. V [1] je provedena analýza podpory bezpečnostních kódů nejdříve v systému Mathematica samotném a následně i v několika jeho rozšířeních od různých autorů. Výsledkem je, že žádné z těchto řešení nesplňuje požadavky na zpracování všech stanovených kódů s dostatečnou názorností pro výukové účely.

V rámci BP proto bylo navrženo a zrealizováno nové řešení kladoucí důraz právě na názornost a kvalitu dokumentace. Zvláště přínosné se ukázaly být dodatečné výpisy jednotlivých funkcí, ve kterých je zachycen průběh daného výpočtu. Systém Mathematica k tomu přispívá například rozsáhlými možnostmi zobrazování matematických výrazů, nástroji na jejich barevné či jiné zvýrazňování nebo funkcemi pro vykreslování tabulek a grafů. Zpracovány tímto způsobem byly kódy: sudá parita, křížová parita, Hammingovy kódy, kódy generované polynomem, prokládané kódy, LDPC kódy a binární BCH kódy. Jejich výběr byl proveden převážně na základě osnovy předmětu *Bezpečnostní kódy* (zkráceně BKO), jehož výuky se účastní Ing. Pavel Kubalík, Ph.D., zadavatel a vedoucí BP i této diplomové práce.

Výstupem BP byla knihovna funkcí a sada výukových scénářů. Knihovnu tvořil adresář s osmi soubory typu balíček (v originále *package*). Každý s těchto

souborů obsahoval funkce příslušející některému z uvedených bezpečnostních kódů. Výukové scénáře pak byly soubory typu notebook, v nichž bylo ukázáno použití jednotlivých funkcí z některého z balíčků. Ukázky byly zároveň doplněny textovým popisem daného bezpečnostního kódu. Přesné členění na jednotlivé soubory zachycuje diagram 1.1.

Balíček je běžný textový soubor, typicky s příponou `.m` nebo `.wl`, obsahující kód v jazyce *Wolfram Language*. Slouží k uložení kódu s minimem nadbytečných informací. Oproti notebookům sice nabízí velmi omezené možnosti formátování, zároveň je však tato podoba výhodnější pro sdílení nebo verzování kódu. Z tohoto důvodu byl v rámci BP pro uložení funkcí použit právě tento typ souboru.



Obrázek 1.1: Výstup bakalářské práce

Bezpečnostní kódy zpracované v BP pokrývají zhruba první polovinu náplně předmětu BKO. V zájmu pokrytí i druhé poloviny obsahu sestávající z pokročilejších bezpečnostních kódů je na BP navázáno touto diplomovou prací. Spadají mezi ně Fireovy kódy, RM kódy, součinové kódy, konvoluční kódy, nebinární BCH kódy a RS kódy. Výukové materiály tohoto předmětu, jejichž autorem je doc. Ing. Alois Pluháček, CSc., zároveň slouží jako hlavní podklady pro implementaci těchto kódů.

Ke kódům z předmětu BKO jsou navíc přidány Goppa kódy zpracované v diplomové práci Ing. Vojtěcha Myslivce [2], jejímž hlavním tématem však není samotný Goppa kód, ale jeho využití v kryptosystému *McEliece*. Autor sice implementoval ukázku kryptosystému i samotných Goppa kódů v systému Mathematica, v některých případech, především u opravy a dekódování, však implementované funkce neposkytují žádné dodatečné informace o průběhu výpočtu. To má být touto prací napraveno.

Implementace nových kódů musí dodržet členění a postupy z BP tak, aby jejich kombinací vznikl jednotný soubor výukových materiálů.

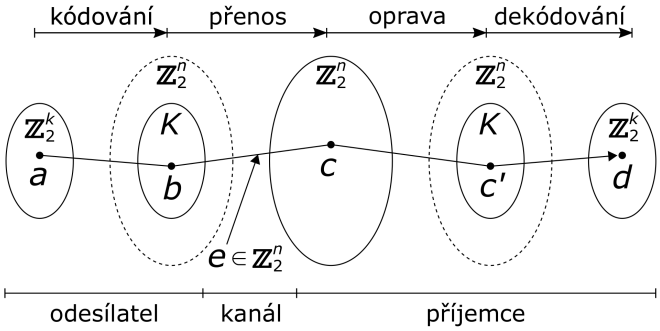
Kapitola 2

Návrh

Tato kapitola je věnována návrhu řešení. Nejprve je v podkapitole 2.1 popsáno, jakým způsobem budou bezpečnostní kódy chápány a jaký to bude mít vliv na vytvořené funkce. Poté je v podkapitole 2.2 uvedeno, co bude hlavním výstupem práce a jak bude tento výstup organizován. Poslední podkapitola 2.3 rozebírá, jaká metodika bude při tvorbě jednotlivých dílčích výstupů použita.

2.1 Pojetí bezpečnostních kódů

Bezpečnostní kódy budou chápány jako sada zobrazení dle schématu 2.1 převzatého z BP. Toto schéma zachycuje přenos zprávy s využitím lineárního binárního bezpečnostního kódu K.



Obrázek 2.1: Schéma přenosu (převzato z [1])

Nejprve je vytvořeno informační slovo a délky k, to je následně kódováním převedeno na slovo kódové b délky n. V průběhu přenosu může dojít k chybě, kterou lze také popsat chybovým slovem e délky n. Přijaté slovo c je pak součtem odeslaného slova b a chybového slova e. Přijaté slovo c je následně opravou převedeno na opravené slovo c' stále délky n, které v ideálním případě

již neobsahuje žádnou chybu. Na závěr je opravené slovo c' převedeno procesem opačným ke kódování, tedy dekódováním, na dekódované slovo d délky k. V ideálním případě je toto identické s původním informačním slovem a.

V maximální možné míře budou v celé práci označována slova v jednotlivých fázích přenosu právě písmeny uvedenými ve schématu a předchozím odstavci. Podobně celá čísla k a n budou v celé práci vyhrazena pro označení dimenze a délky kódu. V některých případech může dekódování probíhat zároveň s opravou, pak bývá celý proces označován také jako dekódování, což může být matoucí. V práci budou tato dvě zobrazení striktně rozlišována.

Výše uvedené schéma slouží k popisu pouze lineárních kódů, tedy takových, že všechna kódová slova tvoří podprostor nějakého vektorového prostoru. Tuto vlastnost ale splňuje většina běžně používaných bezpečnostních kódů včetně těch specifikovaných v zadání práce. Ve schématu jsou uvažovány pouze binární kódy nad $\mathbb{Z}_2$, lze jej však stejně dobře aplikovat i na nebinární kódy nad tělesem $GF(2^m)$ pro $m > 1$.

Každý bezpečnostní kód je zpravidla možné plně určit několika málo parametry, často jimi jsou celá čísla nebo polynomy, na základě kterých jsou vytvořeny další složitější objekty používané při kódování, opravě a dekódování. Typicky jsou to generující a kontrolní matice, nebo generující polynom. Pro každý kód proto bude vytvořena funkce typu `Code`, která ze zadaných parametrů vytvoří všechny objekty nezbytné pro provádění kódování, opravy a dekódování pomocí daného kódu. To lze v pseudokódu vyjádřit jako:

```
K = Code[p1, p2, ...]
```

V souladu se schématem a dřívějším popisem budou pro každý kód definovány funkce typu `Encode`, `Correct` a `Decode` realizující odpovídající operace. Kód `K` vytvořený funkcí typu `Code` tak bude možné předat například funkci typu `Encode` spolu s informačním slovem `a` pro jeho zakódování. Podobně pro opravu a dekódování.

```
b = Encode[K, a]

c' = Correct[K, c]

d = Decode[K, c']
```

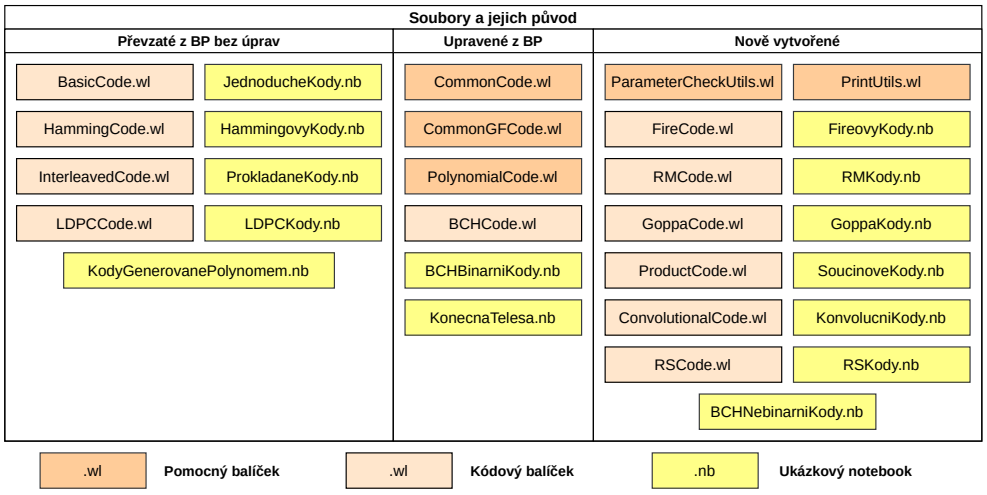
Objekt `K` tedy bude jednoznačně specifikovat daný kód. Jeho použitím se předejde nutnosti předávat jednotlivé parametry `p1`, `p2`, atd. všem funkcím. Zároveň nebude nutné pokaždé generovat všechny objekty potřebné ke kódování, opravě a dekódování ze zadaných parametrů, což může být výpočetně náročné. Budou uloženy jako součást objektu `K` a připraveny k okamžitému použití. Ostatně podobně to platí i při reálném nasazení bezpečnostního kódu. Hardwarová implementace kodéru může být náročná, samotné kódování by pak ale mělo být co nejjednodušší a nejrychlejší.

2.2 Členění na balíčky a ukázkové notebooky

Hlavním výstupem práce bude sada funkcí v programu Wolfram Mathematica pro demonstraci fungování zadaných bezpečnostních kódů. Těmi jsou Fireovy kódy, RM kódy, Goppa kódy, součinové kódy, konvoluční kódy, nebinární BCH kódy a RS kódy.

Tento výstup bude zároveň zkombinován s výstupem BP tak, aby vznikl jednotný celek. V rámci BP byly zpracovány sudá parita, křížová parita, opakovací kód, Hammingovy kódy, kódy generované polynomem, prokládané kódy, LDPC kódy a binární BCH kódy. Funkce příslušející každému kódu jsou umístěny do samostatného souboru typu *package* neboli balíček. Každý balíček je pak doplněn souborem typu notebook s ukázkami použití funkcí z daného balíčku. Toto členění bude dodrženo i pro zpracování nových kódů.

Vedle balíčků věnovaných jednotlivým kódům byly v rámci BP vytvořeny také pomocné balíčky obsahující funkce sdílené několika kódovými balíčky. Při realizaci nových kódů budou jednak využívány existující sdílené funkce a zároveň budou doplněny další. Očekávané úpravy zachycuje diagram 2.2.



Obrázek 2.2: Plánované úpravy existujících materiálů

Pro potřeby nebinárních kódů bude nutné balíčky *CommonCode* a *PolynomialCode* rozšířit o funkce pracující s maticemi, slovy a polynomy nad konečnými tělesy $GF(2^m)$. Balíček *CommonGFCode*, který byl vytvořen právě pro práci s konečnými tělesy $GF(2^m)$, bude pro potřeby nebinárních BCH kódů doplněn o podporu rozšířených těles $GF((2^{m_1})^{m_2})$.

Dále budou přidány dva nové pomocné balíčky. Balíček *PrintUtils* bude sdružovat funkce pro tvorbu dodatečného výpisu, formátování tabulek a jiných výrazů. Do balíčku *ParameterCheckUtils* budou ukládány funkce pro validaci

vstupních parametrů, jak je blíže popsáno v podkapitole 2.3.

Existující balíček *BCHCode* věnovaný binárním BCH kódům bude rozšířen o kódy nebinární. Stávající implementace binárních BCH kódů bude upravena tak, aby dodržovala členění funkcí na konstrukci kódu, kódování, opravu a dekódování. Existující ukázkový notebook věnovaný binárním BCH kódům pokrývá jednak samotné BCH kódy a také konečná tělesa $GF(2^m)$. Notebook bude aktualizován, aby odpovídal přepracovaným binárním BCH kódům, a ukázky práce s konečnými tělesy budou vyčleněny do samostatného notebooku. Nově implementovaným nebinárním BCH kódům bude rovněž věnován samostatný ukázkový notebook.

Pro Fireovy kódy, RM kódy, Goppa kódy, součinové kódy, konvoluční kódy a RS kódy bude vytvořen jak nový balíček, tak nový ukázkový notebook.

Všechny balíčky budou umístěny do adresáře `Packages`, který bude označován jako knihovna. V adresáři `Packages` bude vytvořen vnořený adresář `Internal` pro balíčky *PrintUtils* a *ParameterCheckUtils*, které budou obsahovat funkce určené výhradně pro interní použití v jiných balíčcích. Knihovna bude spolu se všemi ukázkovými notebooky umístěna do adresáře `Reseni`. Ukázkové notebooky budou moci používat funkce z knihovny pouze v případě, že se nachází ve stejném adresáři jako knihovna.

Pro testování funkcí z každého balíčku bude vytvořen testovací notebook, jehož název bude odvozen od jména balíčku přidáním koncovky `Test`. Všechny testovací notebooky budou umístěny do adresáře `Testovani`.

2.3 Metodika tvorby balíčků a notebooků

Při tvorbě nových balíčků a ukázkových notebooků budou dodrženy postupy z BP tak, jak jsou popsány v [1, s. 17–22]. Navíc bude u funkcí prováděna podrobnější kontrola vstupních parametrů. Zatímco funkci `f[x_]` lze zadat jeden libovolný parametr, funkce `f[x_/;IsOk[x]]` je vyhodnocena pouze v případě, že kontrolní funkce `IsOk` pro zadaný výraz vrátí `True`. Bude vytvořena sada takovýchto kontrolních funkcí pro ověřování častých typů parametrů jako jsou binární slova, matice nebo polynomy. Tyto funkce budou tvořit samostatný balíček *ParameterCheckUtils*.

Oproti BP bude nutné pracovat se dvěma novými typy objektů. Jednak to jsou rozšířená konečná tělesa $GF((2^{m_1})^{m_2})$. Ty na rozdíl od běžných těles $GF(2^m)$ nejsou podporovány systémovým balíčkem *FiniteFields*, který prvky tělesa reprezentuje výrazy s hlavičkou `GF`. Vlastní implementace rozšířených těles bude odvozena právě od balíčku *FiniteFields* a pro prvky bude používat výrazy s hlavičkou `EGF`. Související funkce budou umístěny do balíčku *CommonGFCode*. Druhým novým datovým typem jsou objekty sdružující všechny údaje o konkrétním kódu, jak uvádí podkapitola 2.1. Pro ty bude použito asociativní pole, v systému Mathematica označované jako `Association`, tvořené dvojicemi klíč-hodnota. Jako klíče budou používány řetězce.

Kapitola 3

Pomocné balíčky

Vedle balíčků věnovaných jednotlivým kódům, které jsou popsány v kapitole 4, obsahuje knihovna také balíčky funkcí využitelných napříč různými kódy. Právě těmto pomocným balíčkům je věnována následující kapitola. Nejprve jsou v podkapitole 3.1 popsány balíčky převzaté z BP. Informace o dvou nově přidaných pomocných balíčcích jsou pak uvedeny v podkapitole 3.2.

3.1 Převzaté balíčky

Balíčky *CommonCode*, *PolynomialCode* a *CommonGFCode* byly s drobnými úpravami převzaty z BP. Nejdůležitější funkce a případná rozšíření každého z těchto balíčků jsou popsány v následujících třech sekcích.

3.1.1 Balíček *CommonCode*

Původní balíček *CommonCode* obsahoval funkce pro práci s binárními maticemi a slovy. Například pro kódování, opravu a dekódování pomocí matic nebo pro převody mezi kontrolními a generujícími maticemi. Zároveň obsahoval také sadu funkcí pro určování kódové vzdálenosti právě z kontrolní nebo generující matice kódu a řadu dalších pomocných funkcí. Neopomenutelnou součástí balíčku je i pojmenovaný parametr `NoPrint` pro ovlivňování dodatečného výpisu funkcí napříč všemi ostatními balíčky.

Funkce `MatrixEncode` pro zakódování slova pomocí generující matice kódu, `MatrixSyndrome` pro výpočet syndromu přijatého slova z kontrolní matice a `MatrixDecode` pro dekódování přijatého slova generující maticí byly rozšířeny o podporu nebinárních kódů. Tedy kódů, jejichž slova i matice jsou nad tělesem $GF(2^m)$. K realizaci těchto operací byly využity stejné funkce jako u binárních kódů – systémová `Dot` pro maticové násobení a `LinearSolve` pro řešení soustavy lineárních rovnic při dekódování. Podobně byly o možnost práce s nebinárními maticemi rozšířeny funkce `ConvertToG` a `ConvertToH` pro převody z kontrolní matice na generující respektive z generující na kontrolní.

Nejdůležitější funkcí z balíčku *CommonCode* pro manipulaci se slovy je funkce `XorError` používaná pro simulaci přenosu s chybou. Byla rovněž rozšířena o podporu nebinárních slov. Jejím prvním parametrem je odeslané slovo b, tím druhým specifikace chyby. Chyba může být zadána třemi způsoby, buď chybovým slovem stejné délky jako b a nad stejným tělesem nebo pozicemi chyb nebo pouze velikostí chyby. V prvním případě jsou zadaná slova pouze sečtena. Je-li chyba zadána seznamem pozic, jsou v případě binárního slova invertovány odpovídající bity a v případě slova nad $GF(2^m)$ je k prvkům na stanovených pozicích přičten jednotkový prvek daného tělesa. Je-li chyba zadána pouze celým číslem, je v případě binárního slova invertován stanovený počet náhodně vybraných bitů. U slova nad $GF(2^m)$ jsou náhodně voleny nejen pozice chyb, ale i jejich hodnoty.

Příklad 1 *Ukázka použití funkce* `XorError` *pro simulaci přenosu s chybou.*

```
In[ ]:= XorError[{1, 1, 1, 0, 0, 0}, {3, 4}]
Out[ ]= {1, 1, 0, 1, 0, 0}

In[ ]:= XorError[GFFromDecimal[3, {1, 2, 3, 4, 5, 6, 7}], 2] // GFToDecimal
Out[ ]= {2, 2, 7, 4, 5, 6, 7}
```

V prvním případě je slovo binární a chyba je zadána pozicemi. Výsledné slovo má skutečně invertovaný třetí a čtvrtý bit. V druhém případě je slovo nad $GF(2^3)$ zadané v decimální notaci a jako specifikace chyby je funkci `XorError` *předáno celé číslo 2. Pozice chyb i jejich hodnoty jsou tedy voleny náhodně. Výsledek je opět převeden do decimální notace. Vidíme, že v tomto případě chyba zasáhla první a třetí slabiku.*

Jedinou funkcí nově přidanou do balíčku *CommonCode* je `CodeSummary`, která pro libovolný kód reprezentovaný asociací vytvoří jeho shrnutí. Povinnou součástí všech asociací reprezentujících kód je název, délka a dimenze daného kódů spolu se specifikací tělesa, nad nímž jsou jeho slova. Funkce `CodeSummary` vyextrahuje z asociace tyto údaje a zobrazí je přehledně na jednom řádku. Na dalším řádku pak vypíše názvy všech dalších hodnot v asociaci uložených.

Příklad 2 *Ukázka použití funkce* `CodeSummary` *pro shrnutí parametrů kódu.*

Nejprve je definován binární kód K délky $n = 3$ a dimenze $k = 2$ nazvaný jako `"ExampleCode"`*. Součástí popisu kódu je i jeho generující matice G a kontrolní matice H. Tento kód je předán funkci* `CodeSummary`*, která vytvoří shrnutí těchto údajů.*

```
In[•]:= K = <|
          "type" → "ExampleCode", "n" → 3, "k" → 2, "GF" → 2,
          "G" → {{1, 0, 1}, {0, 1, 1}}, "H" → {{1, 1, 1}}
          |>
        CodeSummary[K]
Out[•]= <| type → ExampleCode, n → 3, k → 2, GF → 2, G → {{1, 0, 1}, {0, 1, 1}}, H → {{1, 1, 1}}|>

Out[•]= (3, 2) Example code over GF(2)
            Parameters: type, n, k, GF, G, H
```

3.1.2 Balíček *PolynomialCode*

Balíček původně obsahoval funkce pro práci s polynomy nad $\mathbb{Z}_2$. Většina jich byla rozšířena tak, aby umožňovaly práci i s polynomy nad tělesem $GF(2^m)$.

Funkce z balíčku pokrývají základní operace jako `PolyPlus` pro sčítání polynomů, `PolyTimes` pro jejich násobení, `PolyDivide` pro dělení nebo `PolyMod` pro výpočet zbytku po dělení. Dále jsou v balíčku k dispozici funkce pro zjišťování některých vlastností polynomů, například `PolyDegree` pro zjištění stupně nebo `PolyOrder` pro výpočet řádu. Většina z nich pouze volá některou ze systémových funkcí, umožňují však navíc zadávat polynomy jak symbolicky v proměnné x, tak seznamem koeficientů. Pro převod ze symbolického zápisu na seznam koeficientů slouží funkce `ToList`, pro převod v opačném směru funkce `ToPolynomial`. Pro zobrazování polynomů se hodí funkce `DoubleForm`, která jej zobrazí v obou zmiňovaných formátech.

Obtížnější byla implementace násobení polynomů nad $GF(2^m)$, neboť při jeho provádění pomocí symbolických výpočtů s polynomy v proměnné x stoupala s rostoucími stupni násobených polynomů rapidně také délka výpočtu. Proto bylo násobení ve funkci `PolyTimes` zrealizováno maticovým násobením s polynomy reprezentovanými seznamem koeficientů.

Příklad 3 *Výpočet třetí mocniny polynomu nad $GF(2^3)$ pomocí symbolických výpočtů v proměnné x a maticovým násobením pomocí funkce* `PolyTimes`.

```
In[•]:= (P = ToPolynomial @ GFFromDecimal[3, {1, 2, 3, 4, 5}]) // DoubleForm[GFToDecimal]
Out[•]= 1x^4 + 2x^3 + 3x^2 + 4x + 5  ~  {1, 2, 3, 4, 5}

In[•]:= Expand[P * P * P] // DoubleForm[GFToDecimal] // Timing
Out[•]= {2.09376,
         1x^12 + 2x^11 + 7x^10 + 7x^9 + 7x^8 + 7x^7 + 5x^5 + 1x^4 + 1x^2 + 1x + 6  ~  {1, 2, 7, 7, 7, 7, 0, 5, 1, 0, 1, 1, 6}}

In[•]:= PolyTimes[P, P, P] // DoubleForm[GFToDecimal] // Timing
Out[•]= {0.039588,
         1x^12 + 2x^11 + 7x^10 + 7x^9 + 7x^8 + 7x^7 + 5x^5 + 1x^4 + 1x^2 + 1x + 6  ~  {1, 2, 7, 7, 7, 7, 0, 5, 1, 0, 1, 1, 6}}
```

Polynom P je zadán jako seznam koeficientů nad tělesem $GF(2^3)$ a pomocí funkce `ToPolynomial` *následně převeden na polynom v proměnné x. Zároveň je na tomto příkladu ukázáno použití funkce* `DoubleForm` *pro zobrazování polynomů v obou formách zápisu. Té je navíc předána funkce* `GFToDecimal` *pro*

zobrazování prvků $GF(2^3)$ v decimální notaci. V prvním případě je třetí mocnina P počítána systémovou `Times`*. Pro úplné roznásobení je nutné použít ještě systémovou* `Expand` *a právě v tomto kroku dochází k výraznému zpomalení. Systémová* `Timing` *stanovila dobu běhu na zhruba 2 vteřiny. Zatímco s využitím maticového násobení ve funkci* `PolyTimes` *trvá výpočet pouze 4 setiny vteřiny.*

Násobení polynomů dále využívá například funkce `PolyEncode` provádějící kódování u všech kódů generovaných polynomem. Byla rovněž rozšířena pro práci s nebinárními kódy. Jejím protějškem je funkce `PolyDecode` pro dekódování opraveného slova u všech kódů generovaných polynomem. Tyto funkce jsou velmi často zmiňovány a používány v ukázkách v kapitole 4.

3.1.3 Balíček *CommonGFCode*

Hlavní náplní původního balíčku *CommonGFCode* byly funkce pro práci s konečnými tělesy $GF(2^m)$. Samotné prvky těles jsou sice reprezentovány s využitím systémového balíčku *FiniteFields* a v této podobě s nimi všechny funkce pracují, balíček *CommonGFCode* však umožnil využívat různé formy zápisu pro zadávání těchto prvků a jejich zobrazování. Konkrétně pro zadávání slouží funkce začínající na `GFFrom*` a pro zobrazování funkce začínající na `GFTo*`. Mezi podporované formy zápisu patří například decimální, binární nebo polynomiální. Vedle těchto funkcí obsahoval původní balíček také funkci nazvanou `GFMinimalPolynomial` pro výpočet minimálního polynomu prvku $GF(2^m)$, nebo funkci `GFTable` vytvářející tabulku všech prvků tělesa.

V rámci rozšíření byla do balíčku přidána vlastní implementace rozšířených konečných těles $GF((2^{m_1})^{m_2})$ využívaných především nebinárními BCH kódy. Toto rozšíření si vyžádalo přidání několika funkcí pro hledání ireducibilních polynomů, které vedle nebinárních BCH kódů nachází využití také u Goppa kódů. Zároveň byly modifikovány funkce `GFMinimalPolynomial` a `GFTable` tak, aby rovněž podporovaly rozšířená tělesa. Funkcím z balíčku *CommonGFCode* je věnován samostatný ukázkový notebook `KonecnaTelesa.nb`.

Hledání ireducibilních polynomů

Konečné těleso $GF(2^m)$ je dáno ireducibilním polynomem nad $\mathbb{Z}_2$ stupně m. Podobně rozšířené těleso $GF((2^{m_1})^{m_2})$ je definováno ireducibilním polynomem nad $GF(2^{m_1})$ stupně m_2. Pro ověřování ireducibility obou typů polynomů byla vytvořena funkce `GFIrreduciblePolynomialQ`. Její implementace pro polynomy nad $\mathbb{Z}_2$ je jednoduše voláním systémové `IrreduciblePolynomialQ` s modulem nastaveným na 2. Pro polynomy nad $GF(2^{m_1})$ tento postup nefunguje, musel být implementován vlastní test ireducibility.

Testování ireducibility polynomu nad $GF(2^{m_1})$ stupně m_2 využívá faktu, že polynomy stupně 1 jsou vždy ireducibilní a polynomy stupňů 2 a 3 jsou ireducibilní právě tehdy, když nemají žádný kořen. Neexistence kořenů je nutnou

podmínkou i pro ireducibilitu polynomů stupně $m_2 > 3$, nikoliv však postačující. Pro polynom stupně $m_2 > 1$ jsou proto nejprve hledány jeho kořeny hrubou silou přes všechny prvky $GF(2^{m_1})$. Je-li nějaký kořen nalezen, je jeho ireducibilita vyloučena. Polynomy bez kořenů stupně $m_2 > 3$ jsou následně testovány Rabinovým testem podle [3]. Označíme-li $p_1, \ldots, p_k$ různé prvočíselné dělitele stupně m_2 polynomu f nad $GF(2^{m_1})$, pak f je ireducibilní právě tehdy, když pro všechna $i \in \{1, \ldots, k\}$ platí

$$GCD(f,\ x^{2^{m_1 m_2/p_i}} - x) = 1 \qquad \text{a} \qquad f \text{ dělí } x^{2^{m_1 m_2}} - x\,.$$

Polynomy $x^{2^{m_1 m_2/p_i}}$ a $x^{2^{m_1 m_2}}$ jsou počítány opakovaným mocněním na 2 s redukcí modulo f pomocí systémové `PolynomialRemainder` po každém umocnění. Pro hledání největšího společného násobku byl implementován Euklidův algoritmus, systémová `PolynomialGCD` s polynomy nad konečnými tělesy pracovat neumí.

Příklad 4 *Test ireducibility polynomu nad $GF(2^3)$ stupně 4 pomocí funkce* `GFIrreduciblePolynomialQ`.

```
In[ ]:= P = GFFromDecimal[3, {5, 2, 3, 6, 1}];
        GFIrreduciblePolynomialQ[P, GFToDecimal]

          P(x) = 5x^4 + 2x^3 + 3x^2 + 6x + 1
          deg(P(x)) = 4

        • Checking for roots of P(x)
            → No root found

        • Checking irreducibility using Rabin's test
            → Rabin's test failed

Out[ ]= False
```

Polynom P je zadán seznamem koeficientů z $GF(2^3)$ v decimální notaci. Následně je proveden test jeho ireducibility s negativním výsledkem. Jak je patrné z dodatečného výpisu, polynom sice nemá žádné kořeny, ale neprošel Rabinovým testem.

Test existence kořenů v tomto případě pouze navyšuje dobu běhu. Není dostatečný pro stanovení ireducibility, jelikož polynom je stupně vyššího než 3, a zároveň není ani nutným předstupněm Rabinova testu. U polynomů s kořeny však o jejich reducibilitě může rozhodnout násobně rychleji než Rabinův test.

Jak uvádí [3], zhruba $1/m_2$ polynomů stupně m_2 nad konečným tělesem $GF(2^{m_1})$ je ireducibilních. Lze je tedy hledat opakovaným náhodným výběrem polynomů, dokud není test ireducibility pro některý z nich splněn. To provádí funkce `GFIrreduciblePolynomial`.

Jelikož však hledání ireducibilních polynomů náhodným výběrem může být časově náročné, byl pro každou dvojici $m_1, m_2 \in \{2, \ldots, 10\}$ stanoven výchozí ireducibilní polynom nad $GF(2^{m_1})$ stupně m_2 a zároveň výchozí ireducibilní polynom stupně m_1 nad $\mathbb{Z}_2$ pro definici $GF(2^{m_1})$. Tyto byly vygenerovány v systému *SageMath*. Použitý skript je k nalezení ve složce `Pomocne`. Jeho výstupem je řetězec, který je zároveň validním kódem v jazyce *Wolfram Language*.

Funkce `GFIrreduciblePolynomial` tak ve výchozím nastavení nejprve hledá polynom odpovídající zadaným parametrům mezi výchozími ireducibilními polynomy. Až pokud není nalezen, přistoupí k hledání náhodným výběrem s omezením maximální doby běhu na 30 vteřin. Toto chování lze upravit pojmenovanými parametry `Source` a `TimeLimit`.

Zobrazování prvků konečného tělesa

Prvek $GF(2^m)$ je s využitím balíčku *FiniteFields* reprezentován výrazem s hlavičkou `GF`. Například prvku x^2 z tělesa $GF(2^3)$ daného ireducibilním polynomem $x^3 + x + 1$ odpovídá výraz

```
GF[2, {1, 1, 0, 1}][{0, 0, 1}].
```

Je patrné, že koeficienty polynomů x^2 i $x^3 + x + 1$ jsou zapsány v opačném pořadí, jak je v systému Mathematica zvykem. Balíček *FiniteFields* navíc pro výrazy s hlavičkou `GF` předefinovává systémovou funkci `MakeBoxes` používanou pro zobrazování výrazů v notebooku, takže výše uvedený výraz je zobrazen jako $\{0, 0, 1\}_2$, což může být matoucí pro uživatele zvyklé na zápis koeficientů od nejvyššího členu. V balíčku *CommonGFCode* je proto funkce `MakeBoxes` znovu předefinována na tvar $\{1, 0, 0\}_{GF(8)}$. Aby se předešlo záměně těchto dvou zápisů, je použit jiný formát spodního indexu.

Příklad 5 *Srovnání zobrazovacích formátů prvku x^2 z tělesa $GF(2^3)$ daného ireducibilním polynomem $x^3 + x + 1$.*

```
In[ ]:= << FiniteFields`
        PolynomialToElement[GF[2, {1, 1, 0, 1}], x^2]
Out[ ]= {0, 0, 1}_2

In[ ]:= << CommonGFCode`
        PolynomialToElement[GF[2, {1, 1, 0, 1}], x^2]
Out[ ]= {1, 0, 0}_GF(8)
```

Nejprve je načten pouze balíček FiniteFields a pomocí jedné z jeho funkcí je polynom x^2 převeden na prvek tělesa daného polynomem $x^3 + x + 1$. Prvek je skutečně zobrazen ve formátu popsaném výše.

Následně je načten také balíček CommonGFCode, čímž dojde k předefinování zobrazovací systémové funkce `MakeBoxes` *a stejný prvek je zobrazen s koeficienty v opačném pořadí.*

Rozšířená konečná tělesa

Rozšířené konečné těleso $GF((2^{m_1})^{m_2})$ je tvořeno polynomy nad $GF(2^{m_1})$ stupně $< m_2$. Součtem prvků tělesa je klasický součet polynomů. Součin je prováděn modulo ireducibilní polynom stupně m_2.

Pro reprezentaci prvku $u \in GF((2^{m_1})^{m_2})$ daného ireducibilním polynomem P byl použit výraz s hlavičkou `EGF`, jenž je tvořen seznamem koeficientů prvku u, seznamem koeficientů polynomu P a celým číslem m_1. Pro výrazy s hlavičkou `EGF` jsou předefinovány základní systémové funkce `Plus`, `Times` a `Power`, díky tomu je například možné prvky sčítat operátorem `+` nebo násobit dvě matice nad rozšířeným tělesem standardně systémovou `Dot`. Pro realizaci těchto operací byly použity funkce pro práci s polynomy nad konečnými tělesy z balíčku *PolynomialCode* popsaného výše.

Jelikož výraz `EGF` obsahuje jak koeficienty samotného prvku, tak koeficienty definičního ireducibilního polynomu, může být zápis jediného prvku dosti rozsáhlý. Proto byla pro výrazy s hlavičkou `EGF` taktéž předefinována systémová `MakeBoxes` používaná pro zobrazování výrazů v notebooku. Výsledek ukazuje následující příklad.

Příklad 6 *Vytvoření a reprezentace prvku* $GF((2^3)^2)$.

```
In[ ]:= e = EGFFromGFList[2, GFFromDecimal[3, {1, 7}]]
Out[ ]= {{0, 0, 1}_GF(8), {1, 1, 1}_GF(8)}_GF(8^2)

In[ ]:= InputForm[e]
Out[ ]//InputForm=
EGF[3, {GF[2, {1, 1, 0, 1}][{1, 0, 0}], GF[2, {1, 1, 0, 1}][{0, 1, 0}],
  GF[2, {1, 1, 0, 1}][{0, 1, 0}]}, {GF[2, {1, 1, 0, 1}][{1, 0, 0}],
  GF[2, {1, 1, 0, 1}][{1, 1, 1}]}]
```

Nejprve je pomocí `GFFromDecimal` *vytvořen seznam dvou prvků nad* $GF(2^3)$ *s decimální notací 1 a 7. Následně je tento seznam předán* `EGFFromGFList`*, která z něj udělá prvek* $GF((2^3)^2)$ *s využitím výchozího ireducibilního polynomu nad* $GF(2^3)$ *stupně* 2*. Při jeho zobrazení jsou uplatněny obě modifikace funkce* `MakeBoxes`*, jednak pro prvky* $GF((2^3)^2)$ *a také pro prvky* $GF(2^3)$*. V obou případech je použita notace zápisem koeficientů ve složených závorkách se spodním indexem udávajícím označení daného tělesa.*

V druhé vstupní buňce je pomocí systémové `InputForm` *vypsán kompletní výraz s hlavičkou* `EGF` *odpovídající tomuto prvku. Ten nejprve obsahuje celé číslo* $m_1 = 3$*, následně koeficienty ireducibilního definičního polynomu tělesa* $GF((2^3)^2)$ *a na závěr koeficienty samotného prvku. Pouze modifikací systémové* `MakeBoxes` *je dosaženo toho, že je tento výraz při vypsání převeden do stručnějšího formátu.*

Minimální polynomy

Minimální polynomy jsou zásadní při konstrukci binárních a nebinárních BCH kódů. U funkce `GFMinimalPolynomial` bylo jednak umožněno zpracovávat prvky rozšířených těles a zároveň byl také obohacen její dodatečný výpis.

Minimálním polynomem prvku $\beta \in GF(2^m)$ je polynom $M_\beta(x)$ nad $\mathbb{Z}_2$ nejmenšího stupně takový, že jeho vedoucí koeficient je roven 1 a $M_\beta(\beta) = 0$. Je-li $i \in \mathbb{N}$ nejmenší takové, že $\beta^{2^i} = \beta$, pak

$$M_\beta(x) = (x+\beta)\,(x+\beta^2)\,(x+\beta^{2^2})\,\ldots\,(x+\beta^{2^{i-1}})\,.$$

Minimálním polynomem prvku $\beta \in GF((2^{m_1})^{m_2})$ je polynom $M_\beta(x)$ nad $GF(2^{m_1})$ nejmenšího stupně takový, že jeho vedoucí koeficient je roven 1 a $M_\beta(\beta) = 0$. Je-li $q = 2^{m_1}$ a $i \in \mathbb{N}$ nejmenší takové, že $\beta^{q^i} = \beta$, pak

$$M_\beta(x) = (x+\beta)\,(x+\beta^q)\,(x+\beta^{q^2})\,\ldots\,(x+\beta^{q^{i-1}})\,.$$

V obou případech jsou minimální polynomy hledány tímto postupem. K nalezení odpovídajícího i je použita systémová `NestWhile`, která opakovaně aplikuje tutéž funkci na její výstup z předchozího kroku, dokud není splněna jistá podmínka. Platí totiž

$$\beta^{2^i} = (\beta^{2^{i-1}})^2 \qquad \text{a} \qquad \beta^{q^i} = (\beta^{q^{i-1}})^q\,.$$

Příklad 7 *Minimální polynom prvku rozšířeného tělesa $GF((2^3)^2)$.*

```
In[ ]:= e = EGFFromGFList[2, GFFromDecimal[3, {1, 7}]];
        GFMinimalPolynomial[e, GFToBinaryString]
```

β = {001, 111}$_{GF(8^2)}$

i	8^i	β^{8^i}
0	1	{001, 111}$_{GF(8^2)}$
1	8	{001, 101}$_{GF(8^2)}$
2	64	{001, 111}$_{GF(8^2)}$

$M_\beta(x) = (x + \beta^1)(x + \beta^8)$

$M_\beta(x) = (x + \{001, 111\}_{GF(8^2)})(x + \{001, 101\}_{GF(8^2)})$

$M_\beta(x) = \{0, 001\}_{GF(8^2)}\, x^2 + \{0, 010\}_{GF(8^2)}\, x + \{0, 100\}_{GF(8^2)}$

$M_\beta(x) = 001\, x^2 + 010\, x + 100$

```
Out[ ]= {1, 0, 0}_GF(8) + x {0, 1, 0}_GF(8) + x^2 {0, 0, 1}_GF(8)
```

Nejprve je pomocí `GFFromDecimal` *vytvořen seznam dvou prvků nad $GF(2^3)$ s decimální notací 1 a 7. Následně je tento seznam předán* `EGFFromGFList`*, která z něj udělá prvek $GF((2^3)^2)$ s využitím výchozího ireducibilního polynomu nad $GF(2^3)$ stupně 2. Tento prvek je následně předán* `GFMinimalPolynomial` *spolu s funkcí* `GFToBinaryString` *způsobující zobrazení prvků $GF(2^3)$ binárními řetězci.*

Tabulka zachycuje hledání minimálního $i \geq 1$ takového, že $\beta = \beta^{8^i}$, neboť $8 = 2^3$ je počet prvků vnitřního tělesa. Následně je dopočten výsledný minimální polynom. I když je β i všechny jeho mocniny z $GF((2^3)^2)$, součinem vznikne polynom s koeficienty z $GF((2^3)^2)$ takovými, že každý z nich je konstantním polynomem nad $GF(2^3)$, a tedy i prvkem $GF(2^3)$.

3.2 Nově vytvořené balíčky

Dvojice nově vytvořených pomocných balíčků *ParameterCheckUtils* a *PrintUtils* je popsána v následujících dvou sekcích. Tyto balíčky jsou považovány za interní, neočekává se, že by je uživatel používal přímo.

3.2.1 Balíček *ParameterCheckUtils*

Balíček *ParameterCheckUtils* slouží ke kontrole vstupních parametrů funkcí. Balíčky věnované jednotlivým kódům zpravidla obsahují pouze nízký počet funkcí, zato je však možné je volat s různým počtem argumentů a s argumenty zadanými v různých formátech. Typicky polynomy je možné zadávat seznamem koeficientů nebo symbolicky polynomem v proměnné x. Snadno tak může dojít k situaci, kdy jsou některé z argumentů zadány chybně.

Pomocí funkcí z balíčku *ParametrCheckUtils* je umožněno kontrolovat nejběžnější typy parametrů ještě před samotným spuštěním hlavní funkce a upozornit uživatele na konkrétní nedostatek v jeho zadání. Běžnými typy parametrů u funkcí věnovaných bezpečnostním kódům jsou například binární slova, binární matice, celá čísla jisté minimální velikosti, slova nad tělesy $GF(2^m)$, polynomy, funkce nebo asociace reprezentující kód.

Pro každý z těchto a dalších dohromady více než 20 typů je v balíčku dedikovaná funkce, která rozhodne, zdali je jí předaný výraz daného typu, či ne, a vrátí odpovídající hodnotu `True`, nebo `False`. Každá funkce navíc existuje ve dvou verzích. Jedna pouze vrátí odpovídající výsledek, druhá v případě neúspěchu navíc vypíše chybovou hlášku. Varianta s hláškou je nazvána připojením `M` za název funkce bez hlášky.

Příklad 1 *Ověření binárního slova funkcemi* `IsWord2` *a* `IsWord2M`*.*

```
In[ ]:= {IsWord2[{1, 0, 1, 0}], IsWord2[{1, 0, 1, 2}]}
Out[ ]= {True, False}

In[ ]:= {IsWord2M[{1, 0, 1, 0}], IsWord2M[{1, 0, 1, 2}]}
        ParameterCheck: Cannot interpret expression {1, 0, 1, 2} as a word over GF(2).
Out[ ]= {True, False}
```

Nejprve je funkci `IsWord2` *provádějící test na binární slovo předáno k ověření slovo 1010 a následně slovo 1012. V prvním případě vrátí dle očekávání*

hodnotu `True` *a ve druhém* `False`*. Odlišné chování funkce* `IsWord2M` *je vidět vzápětí na vypsané chybové hlášce, jinak je její výstup totožný s* `IsWord2`*.*

Vždy bude existovat chybný vstup, který funkcemi z balíčku *ParameterCheckUtils* nebude zachycen. Funkce nejsou navrženy ke stoprocentnímu pokrytí všech případů, měly by však postačit k odhalení chybných vstupů způsobených nepozorností. V některých případech může mít testování parametrů negativní vliv na délku běhu výpočtu. Prioritou však nebyla rychlost, ale názornost a jednoduchost použití.

3.2.2 Balíček *PrintUtils*

Balíček *PrintUtils* sdružuje funkce používané pro tvorbu dodatečného výpisu. Nejdůležitější z nich je funkce `Printer`. Umožňuje snadno vytvářet dodatečný výstup tak, aby byl všechen součástí jediné buňku. Opakované volání systémové `Print` totiž pokaždé vytvoří buňku novou, což může být při rozsáhlém dodatečném výpisu nepřehledné. Navíc je u každého takového volání funkce `Print` nutné testovat, zdali není dodatečný výpis zakázán. Preferovaný postup je tedy zavolat `Print` pouze jednou až na konci funkce, kdy je všechen obsah připraven.

Funkce `Printer` vrátí funkci, kterou lze používat stejně jako systémovou `Print` až na to, že samotný výstup je vytvořen až ve chvíli, kdy je tato funkce zavolána bez jakýchkoli argumentů. Navíc se tato funkce automaticky řídí hodnotou parametru `NoPrint` používaného napříč všemi balíčky k zakázání dodatečného výpisu.

Příklad 2 *Ukázka použití funkce* `Printer`*.*

```
In[ ]:= pr = Printer[NoPrint → False];
        pr["A", "B", "C", "\n"];
        pr["1+2 = "];
        pr[3];
        pr[];
        "Output"

        ABC
        1+2 = 3

Out[ ]= Output
```

Funkci `Printer` *je předán pojmenovaný parametr* `NoPrint` *nastavený na* `False`*, dodatečný výpis je tedy povolen. V tom případě vytvoří funkce* `Printer` *novou unikátní globální proměnnou pomocí systémové* `Unique` *a inicializuje ji na prázdný seznam. Zároveň vrátí funkci, zde pojmenovanou jako* `pr`*, pomocí níž je do tohoto seznamu možné přidávat položky. Je-li pak* `pr` *zavolána bez argumentů, je celý tento seznam vypsán systémovou* `Print` *a daná globální proměnná uvolněna.*

Ačkoliv to z obrázku není přímo patrné, řádky výstupu začínající na `ABC` *a* `1+2` *jsou součástí téže buňky. Řetězec* `"Output"` *simuluje skutečnou návratovou hodnotu nějaké funkce, je umístěn do samostatné buňky pod dodatečný výpis.*

Další často používanou funkcí je `GrayTable` pro formátování tabulek. Ta interně volá systémovou `Grid`. Zadává u ní však řadu dodatečných parametrů ovlivňujících například barvu jednotlivých řádků nebo způsob jejich ohraničení. Navíc umožňuje pohodlně zadávat textové nadpisy sloupců nezávisle na datech.

Příklad 3 *Ukázka použití funkce* `GrayTable` *pro formátování tabulkových dat.*

```
In[ ]:= GrayTable[{"Column 1", "Column 2"}, {{1, 2}, {3, 4}, {5, 6}}]
```

Out[ ]=

Column 1	Column 2
1	2
3	4
5	6

Funkci jsou nejprve předány nadpisy sloupců, následně data v podobě matice. Záhlaví tabulky má tmavší pozadí. Řádky jsou následně střídavě podbarvovány bílou a světle šedou.

Jak již bylo zmíněno dříve Mathematica zobrazuje polynomy od nejnižšího členu po nejvyšší. Navíc jsou-li koeficienty polynomu z konečného tělesa $GF(2^m)$, nedokáže je správně seřadit. Pro zobrazování polynomů proto byly vytvořeny funkce `PolyRow` a `PolyStr`. Zatímco první jmenovaná využívá pro zobrazení polynomu systémovou `Row`, druhá vytvoří z polynomu jeden řetězec.

Příklad 4 *Ukázka použití funkce* `PolyRow` *pro formátování polynomů nad konečným tělesem.*

```
In[ ]:= P = ToPolynomial@GFFromDecimal[3, {1, 0, 3, 4, 5, 0, 7}]
Out[ ]= x^3 {1, 0, 0}_GF(8) + x^6 {0, 0, 1}_GF(8) + x^2 {1, 0, 1}_GF(8) + x^4 {0, 1, 1}_GF(8) + {1, 1, 1}_GF(8)

In[ ]:= PolyRow[P, GFToBinaryString]
Out[ ]= 001x^6 + 011x^4 + 100x^3 + 101x^2 + 111
```

Polynom P je zadán jako seznam koeficientů z $GF(2^3)$. Poté je převeden na polynom v proměnné x. Mathematica však neumí jednotlivé monomy správně seřadit. Například nejvyšší člen x^6 je v pořadí jako druhý.

Následně je tento polynom zformátován funkcí `PolyRow`*. Je jí navíc předána funkce* `GFToBinaryString`*, pomocí níž budou naformátovány jednotlivé koeficienty z $GF(2^m)$. V tomto případě jsou výsledkem binární řetězce.*

Vedle výše demonstrovaných funkcí obsahuje balíček *PrintUtils* ještě celou řadu jednoduchých funkcí pro výpis řetězců s různými kombinacemi horních a dolních indexů před a za slovem, navíc s možností zobrazení některého z nich kurzívou. Interně používají především systémové `Superscript`, `Subscript` a `Subsuperscript`.

Kapitola 4

Bezpečnostní kódy

Tato kapitola popisuje jednotlivé zpracované bezpečnostní kódy a zároveň i jejich implementaci. Každá podkapitola je věnována jednomu kódu, případně některé jeho variantě. Pro každý kód je nejprve popsána konstrukce kódu, následně kódování a na závěr oprava a dekódování. Každá tato část začíná matematickým popisem, který je však vzápětí doplněn popisem odpovídajících implementovaných funkcí a ukázkami jejich použití. Ukázky doplňují matematický popis a zároveň k němu často odkazují, proto jsou tyto umístěny poblíž sebe, nikoliv do odlišných kapitol.

4.1 Fireovy kódy

Fireovy kódy jsou binární kódy generované polynomem pro opravu jednoho shluku chyb určité délky. Související funkce jsou implementovány v balíčku *FireCode*, ukázky jejich použití se nachází v notebooku `FireovyKody.nb`.

Nejprve je v sekci 4.1.1 popsána konstrukce kódu, následně v 4.1.2 kódování a na závěr v 4.1.3 oprava a dekódování.

Informace čerpány především z [4], [5], [6], dále z [7, s. 261–269].

4.1.1 Konstrukce kódu

Generující polynom Fireova kódu $G(x) \in \mathbb{Z}_2[x]$ je součinem dvou polynomů

$$G(x) = P(x) \cdot Q(x)\,,$$

kde $P(x)$ je ireducibilní polynom a $Q(x) = x^q + 1$ pro nějaké $q \in \mathbb{N}$. Stupeň $P(x)$ je minimálně 2, označme jej deg_P. Další důležitou vlastností $P(x)$ je jeho řád, tedy nejmenší $ord_P \in \mathbb{N}$ takové, že $P(x)$ dělí polynom $x^{ord_P} + 1$. Aby se jednalo o Fireův kód musí být mezi ord_P a q jistý vztah. Různé zdroje se však v požadavcích na vztah ord_P s q rozcházejí.

Dle [4] musí být ord_P a q nesoudělné. Dle [7, s. 261] musí být q liché a nedělitelné ord_P, navíc musí platit $q < 2\,deg_P$. Některé kódy splňují obě definice, některé jen tu první a některé jen tu druhou.

Pro délku kódu platí

$$n = LCM(ord_P,\, q)\,,$$

je tedy rovna nejmenšímu společnému násobku čísel ord_P a q. V případě první definice, kdy je zaručena nesoudělnost těchto dvou, lze psát jednoduše $n = ord_P \cdot q$. Redundance kódu r je jako u ostatních kódů generovaných polynomem rovna stupni generujícího polynomu. Jelikož je $G(x)$ součinem $P(x)$ a $Q(x)$, získáváme $r = deg_P + q$. A pro dimenzi kódů

$$k \quad = \quad n - r \quad = \quad n - deg_P - q\,.$$

Pomocí Fireova kódu je možné detekovat shluk chyb do délky L_D a opravit shluk chyb do délky L_C, pokud

$$L_C \leq L_D \quad \wedge \quad L_C + L_D \leq q + 1 \quad \wedge \quad L_C \leq deg_P\,.$$

Tato práce uvažuje pouze korekční schopnosti kódů. Z prvních dvou nerovností plyne $2L_C \leq q+1$. Pro maximální opravu tak v kombinaci s poslední nerovností $L_C \leq deg_P$ dostáváme

$$\begin{aligned} L_C &= \min\left(deg_P,\, \lfloor (q+1)/2 \rfloor\right), \\ L_D &= (q+1) - L_C\,. \end{aligned}$$

Máme-li pevně daný ireducibilní polynom $P(x) \in \mathbb{Z}_2[x]$ s odpovídajícími deg_P a ord_P a měníme-li přitom hodnotu q, zjistíme při použití první definice, že s rostoucím q plynule stoupá detekční schopnost kódů L_D a zároveň dochází k navyšování délky kódu n a jeho dimenze k. Ze vztahu pro L_C je však patrné, že korekční schopnost kódu od jisté úrovně již dále neroste. Stane se tak ve chvíli, kdy q je zhruba rovno $2\,deg_P$. Tedy když $Q(x)$ je zhruba dvojnásobného stupně oproti $P(x)$.

U druhé definice je nárůst q omezen vztahem $q < 2\,deg_P$. Tudíž není možné aby detekční schopnost kódu L_D výrazně převyšovala jeho korekční schopnost L_C. Dokonce z omezení hodnoty q dostáváme $(q+1)/2 \leq deg_P$ a tedy $L_C = \lfloor (q+1)/2 \rfloor$. Při zahrnutí omezení na q lichá pak pro kódy dle druhé definice platí

$$L_C = L_D = (q+1)/2\,.$$

Dalším rozdílem oproti první definici je to, že délka kódu n nemusí s rostoucím q vždy růst. To je dáno připuštěním q soudělného s ord_P.

Implementace

Konstrukci Fireova kódu provádí funkce `FireCode` se dvěma pozičními parametry – polynomem $P(x)$ a exponentem q z polynomu $Q(x) = x^q + 1$. Funkce nejprve ověří, zdali zadané hodnoty skutečně splňují požadavky z definice, například ireducibilitu $P(x)$, či vztahy mezi q a řádem a stupněm $P(x)$. Pojmenovaným parametrem `QCriterion` je možné nastavit, která z definic má být kontrolována. Její hodnotou je řetězec. První definice dle [4] je brána jako výchozí, odpovídá jí řetězec `"Coprime"`, jelikož tato definice vyžaduje nesoudělnost q a ord_P. Druhou definici je možné vynutit nastavením `QCriterion` na `"OddNotDivisible"`, protože požaduje q liché a nedělitelné ord_P.

Kontrola ireducibility je provedena systémovou `IrreduciblePolynomialQ` s pojmenovaným parametrem `Modulus` nastaveným na `2`. K výpočtu řádu $P(x)$ je použita `PolyOrder` z balíčku *PolynomialCode* provádějící postupné navyšování exponentu u v polynomu $x^u + 1$, dokud tento není dělitelný $P(x)$. K výpočtu zbytku po dělení je použita `PolyMod` opět z *PolynomialCode*.

Samotná konstrukce kódu je po provedení kontroly definice velmi jednoduchá. Stačí z parametru q vytvořit polynom $Q(x)$ a ten pak pomocí `PolyTimes` z balíčku *PolynomialCode* vynásobit $P(x)$. Výpočet parametrů kódu využívá pouze jednoduché matematické operace dle vzorců výše.

Příklad 1 *Konstrukce Fireova kódu dle první definice s polynomy* $P(x) = x^3 + x + 1$ *a* $Q(x) = x^5 + 1$.

```
In[ ]:= FireCode[x^3 + x + 1, 5, QCriterion → "Coprime"]

        P(x) = x^3 + x + 1    →    deg_P = 3, ord_P = 7
        Q(x) = x^5 + 1    →    q = 5
        G(x) = P(x)·Q(x) = x^8 + x^6 + x^5 + x^3 + x + 1

        n = q · ord_P = 5 · 7 = 35
        k = n − deg_P − q = 35 − 3 − 5 = 27

        L_C = min[deg_P, ⌊(q + 1) / 2⌋] = min[3, 3] = 3
        L_D = q + 1 − L_C = 5 + 1 − 3 = 3

Out[ ]= ⟨| type → FireCode, n → 35, k → 27, GF → 2, L_C → 3, L_D → 3, P(x) → 1 + x + x^3,
          Q(x) → 1 + x^5, G(x) → 1 + x + x^3 + x^5 + x^6 + x^8, q → 5, EC → runs of 3 errors |⟩
```

Funkci je předán polynom $P(x)$ zapsaný symbolicky v proměnné x, exponent q polynomu $Q(x)$ a pojmenovaný parametr určující, která z definic má být použita.

Výstupem funkce je kód reprezentovaný asociací. Mezi parametry kódu patří jeho typ, délka n, dimenze k, délka opravitelného shluku L_C, délka detekovatelného shluku L_D, polynomy $G(x)$, $P(x)$ a $Q(x)$ spolu s exponentem q. Nechybí řetězec EC pro shrnutí korekční schopnosti daného kódu. Dodatečný výpis dokumentuje postup stanovení těchto parametrů.

Hledání validní kombinace polynomu $P(x)$ a exponentu q je usnadněno tím, že je možné funkci `FireCode` zadat pouze $P(x)$ a platná q jsou nalezena automaticky a zobrazena přehledně v tabulce. Dokonce není nutné ani znát nějaký ireducibilní polynom, místo $P(x)$ je možné funkci předat celé číslo a funkce projde všechny ireducibilní polynomy daného stupně. Ty jsou hledány hrubou silou testováním všech polynomů daného stupně na ireducibilitu pomocí systémové `IrreduciblePolynomialQ`.

Pojmenovaným parametrem `QCriterion` je stále možné ovlivnit výběr definice. Je-li nastaven na `"Both"`, je umožněno srovnání obou definic. Dalším pojmenovaným parametrem je `QMaxValue`, pomocí nějž lze nastavit maximální hodnotu q. Jeho výchozí hodnotou je klíčové slovo `Automatic`, které způsobí automatické stanovení maxima q tak, aby k danému $P(x)$ byl nalezen kód s maximální korekční schopností L_C. Jinak může být uživatelem nastaven na libovolné přirozené číslo.

Příklad 2 *Přehled Fireových kódu s polynomem $P(x) = x^4 + x + 1$ nabízející srovnání obou definic.*

```
In[•]:= FireCode[x^4 + x + 1, QCriterion → "Both", QMaxValue → 15]
```

P(x) = x^4 + x + 1

ord[P(x)] = 15

Coprime			q	OddNotDivisible		
L_0	L_D	(n,k)		L_0	L_D	(n,k)
4	11	{210, 192}	14	–	–	–
4	10	{195, 178}	13	–	–	–
4	8	{165, 150}	11	–	–	–
4	5	{120, 108}	8	–	–	–
4	4	{105, 94}	7	4	4	{105, 94}
–	–	–	5	3	3	{15, 6}
2	3	{60, 52}	4	–	–	–
–	–	–	3	2	2	{15, 8}
1	2	{30, 24}	2	–	–	–
1	1	{15, 10}	1	1	1	{15, 10}

Funkci je předán polynom $P(x)$, požadavek na zahrnutí kódů dle obou definic a maximální hodnota q.

V levé části výstupu jsou uvedeny informace o $P(x)$. Tabulka pak v prostředním sloupci obsahuje hodnotu q, v levé části parametry kódu dle první definice, pokud pro danou hodnotu q kód existuje. A v pravé části parametry kódu dle druhé definice. Mezi údaji o kódu je zahrnuta korekční a detekční schopnost kódu (sloupce L_C a L_D), jeho délka a dimenze (sloupce n a k). Při umístění kurzoru nad hodnoty n a k je zobrazen jejich poměr.

4.1.2 Kódování

Kódování u Fireova kódu je prováděno jako u ostatních kódů generovaných polynomem. A to tak, že je informační slovo $a \in \mathbb{Z}_2^k$ považováno za seznam koeficientů polynomu $A(x)$, který je vynásoben generujícím polynomem $G(x)$ stupně r. Kódové slovo $b \in \mathbb{Z}_2^n$ je seznam koeficientů polynomu

$$B(x) = A(x) \cdot G(x)\,.$$

Takto provedené kódování není systematické, tedy prvních k bitů b není identické slovu a. Lze však dosáhnout i systematického kódování při zachování skutečnosti, že každé kódové slovo $B(x)$ je násobkem $G(x)$. Předpis pro kódování je pak

$$B(x) = A(x) \cdot x^r + (A(x) \cdot x^r) \,\%\, G(x)\,,$$

kde % značí operaci modulo polynom.

Implementace

Jelikož je proces kódování společný všem kódům generovaným polynomem, není realizován samostatnou funkcí v balíčku *FireCode*. Ke kódování je nutné využít balíčku *PolynomialCode*, konkrétně funkce `PolyEncode`.

Příklad 3 *Systematické zakódování informačního slova Fireovým kódem s polynomy* $P(x) = x^3 + x + 1$ *a* $Q(x) = x^3 + 1$.

```
In[ ]:= K = FireCode[x^3 + x + 1, 3, NoPrint → True];
        wordA = {0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0};
        PolyEncode[K["G(x)"], wordA, ListLength → K["n"], CodeType → "Systematic"]
            B(x) = A(x)·x^r + (A(x)·x^r) % G(x)          where r = deg[G(x)]
Out[ ]= {0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1}
```

Nejprve je definován kód K*, následně informační slovo* a*. Toto slovo je zakódováno pomocí generujícího polynomu* $G(x)$ *kódu* K*, který je jedním z klíčů v asociaci reprezentující kód. Pojmenovaný parametr* `ListLenght` *udává délku výsledného slova jakožto seznamu koeficientů polynomu* $B(x)$*. Parametr* `CodeType` *nastavuje typ kódování na systematické. V rámci dodatečného výpisu je zobrazen předpis kódování.*

4.1.3 Oprava a dekódování

Oprava Fireových kódů využívá faktu, že jsou všechny tyto kódy cyklické. Tedy že libovolnou rotací kódového slova dostaneme opět slovo kódové. Dále využívá to, že generující polynom $G(x)$ je součinem dvou polynomů $P(x)$ a $Q(x)$.

Mějme Fireův kód délky n pro opravu shluku L_C chyb. Označíme $G_1(x)$ ten z dvojice polynomů $P(x)$ a $Q(x)$, který je vyššího stupně. V případě rovnosti dostane přednost $Q(x)$. Zbylý polynom označíme $G_2(x)$. Postup opravy přijatého slova $c \in \mathbb{Z}_2^n$, které je interpretováno jako polynom $C(x)$ je následující:

1. Nalezení nejmenšího $u \in \mathbb{N}_0$ takového, že polynom $(C(x) \cdot x^u) \,\%\, G_1(x)$ je stupně menšího než L_C.
2. Označení $F(x) = (C(x) \cdot x^u) \,\%\, G_1(x)$.

3. Nalezení nejmenšího $v \in \mathbb{N}_0$ takového, že polynom $(C(x) \cdot x^v) \,\%\, G_2(x)$ je roven $F(x)$.

4. Vyřešení soustavy kongruencí pro ξ:

$$\xi \equiv u \pmod{ord_{G_1}} \quad \wedge \quad \xi \equiv v \pmod{ord_{G_2}},$$

kde ord_{G_1} a ord_{G_2} značí řády daných polynomů.

5. Kombinace $F(x)$ a ξ ke stanovení chybového slova $E(x)$:

$$E(x) = F(x) \cdot x^{(n-\xi)\,\%\,n}.$$

Neboli $F(x)$ udává tvar shluku chyb a ξ jeho pozici. Oprava je dokončena standardně odečtením chybového slova $C'(x) = C(x) - E(x)$.

Symbol % značí operaci modulo. V exponentu $x^{(n-\xi)\,\%\,n}$ v bodě 5 je operace modulo pouze pro případ $\xi = 0$, tedy chyba na posledním bitu, pro žádné jiné ξ redukce neproběhne. Vyjdou-li jak u, tak v nulové, nebyla detekována žádná chyba a opravu je možné ukončit.

Alternativně lze použít obdobný postup, který je společný všem cyklickým kódům pro opravu shluků chyb a využívá pouze polynomu $G(x)$:

1. Nalezení nejmenšího $\xi \in \mathbb{N}_0$ takového, že $(C(x) \cdot x^{\xi}) \,\%\, G(x)$ je stupně menšího než L_C.

2. Označení $F(x) = (C(x) \cdot x^{\xi}) \,\%\, G(x)$

3. Kombinace $F(x)$ a ξ ke stanovení chybového slova $E(x)$:

$$E(x) = F(x) \cdot x^{(n-\xi)\,\%\,n}.$$

Tento postup je označován jako Meggittův dekodér. Na Fireových kódech však může vyžadovat více kroků než první varianta s dvojicí polynomů $G_1(x)$ a $G_2(x)$ a soustavou kongruencí.

Postup dekódování opraveného slova $C'(x)$ je stejně jako kódování společný s ostatními kódy generovanými polynomem a existuje ve dvou variantách pro systematický a nesystematický kód. Dekódované slovo $d \in \mathbb{Z}_2^k$ je seznamem koeficientů polynomu

$$D(x) = C'(x) \,/\, G(x),$$

v případě nesystematického kódu. V případě systematického, označíme-li stupeň $G(x)$ jako r, platí

$$D(x) = C'(x) \,/\, x^r.$$

Operace / značí dělení polynomů. V obou případech je za předpokladu správné opravy zaručeno, že $C'(x)$ je násobkem $G(x)$. U nesystematického dekódování tudíž dělení proběhne beze zbytku. U systematického dekódování není zaručeno, že $C'(x)$ je násobkem x^r, zbytek po dělení je však ignorován. V podstatě jde o zahození nejnižších koeficientů $C'(x)$, které odpovídají paritním bitům.

Implementace

Funkce `FireCorrect` provádí opravu dle prvního zmíněného postupu. Vedle řady již dříve uvedených operací nad polynomy z balíčku *PolynomialCode* tak musí řešit soustavu kongruencí. To je realizováno pomocí čínské věty o zbytcích, přesněji její implementace v podobě systémové `ChineseRemainder`.

Příklad 4 *Oprava přijatého slova Fireova kódu s polynomy* $P(x) = x^3 + x + 1$ *a* $Q(x) = x^3 + 1$.

```
In[•]:= K = FireCode[x^3 + x + 1, 3, NoPrint → True];
        wordC = {0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1};
        FireCorrect[K, wordC]

        P(x) = x^3 + x + 1, Q(x) = x^3 + 1, n = 21, L_C = 2

        deg[P(x)] ≤ deg[Q(x)]   →   G_1(x) = Q(x) = x^3 + 1
                                    G_2(x) = P(x) = x^3 + x + 1

        C(x) = x^19 + x^17 + x^15 + x^11 + x^10 + x^9 + x^4 + x^3 + x^2 + 1

        1. Find smallest u such that: deg[C(x)·x^u % G_1(x)] < 2
        2. Let F(x) = C(x)·x^u % G_1(x)
        3. Find smallest v such that: C(x)·x^v % G_2(x) = F(x)
```

u/v	C(x)·x^u % G_1(x)	C(x)·x^v % G_2(x)
0	x^2 + x	x^2 + x + 1
1	x^2 + 1	x^2 + 1
2	x + 1	1
3	–	x
4	–	x^2
5	–	x + 1

```
                                                    →   u = 2
                                                        F(x) = x + 1
                                                        v = 5

        4. Solve system of congruences:
            ξ ≡ u (mod ord[G_1(x)])          ξ ≡ 2 (mod 3)
                                        →                       →   ξ = 5
            ξ ≡ v (mod ord[G_2(x)])          ξ ≡ 5 (mod 7)
        5. Let E(x) = F(x)·x^((n−ξ)%n)   →   E(x) = (x + 1)·x^((21 − 5) % 21) = (x + 1)·x^16

Out[•]= {0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1}
```

Nejprve je definován kód K *a přijaté slovo* c *s chybou na čtvrtém a pátém bitu. Tyto dva údaje jsou předány jako argumenty funkci* `FireCorrect`*. Jejím výstupem je opravené slovo* c'*. Dodatečný výpis obsahuje nejprve základní informace o použitém kódu, přijatém slovu a určení, který z polynomů* $P(x)$, $Q(x)$ *je* $G_1(x)$ *a který* $G_2(x)$*. Následně rozepisuje opravu do pěti kroků, jako v popisu výše. První 3 kroky jsou shrnuty do společné tabulky obsahující jednotlivé zbytky po dělení.*

Pro dekódování není v balíčku *FireCode* dedikovaná funkce. Používá se `PolyDecode` z balíčku *PolynomialCode* jakožto protějšek `PolyEncode` zmiňované v sekci o kódování.

Příklad 5 *Systematické dekódování opraveného slova Fireova kódu s polynomy* $P(x) = x^3 + x + 1$ *a* $Q(x) = x^3 + 1$.

```
In[ ]:= K = FireCode[x^3 + x + 1, 3, NoPrint → True];
        wordCCor = {0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1};
        PolyDecode[K["G(x)"], wordCCor, ListLength → K["k"], CodeType → "Systematic"]

            D(x) = C'(x) % x^r          where r = deg[G(x)]

Out[ ]= {0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0}
```

Opět je nejprve vytvořen kód K*, poté opravené slovo* c'*. Funkci* `PolyDecode` *je předán generující polynom kódu* K*, slovo* c'*, pojmenovaný parametr určující délku dekódovaného slova a další pojmenovaný parametr určující typ dekódování. Výstupem je dekódované slovo* d*.*

4.2 RM kódy

Reed-Mullerovy (zkráceně RM) kódy jsou binární kódy generované maticí pro opravu volitelného množství chyb. Vyznačují se snadno implementovatelnou metodou opravy a dekódování. Související funkce jsou implementovány v balíčku *RMCode* a ukázky jejich použití se nachází v notebooku `RMKody.nb`.

Nejprve je v sekci 4.2.1 popsána konstrukce kódu, následně v 4.2.2 kódování a na závěr v 4.2.3 oprava a dekódování.

Informace čerpány z [8] a [9, s. 81–97].

4.2.1 Konstrukce kódu

Při konstrukci RM kódů se využívá jistého typu logických funkcí – tzv. prostých součinů. Je-li $\mu \in \mathbb{N}$, pak logickou funkci $f : \mathbb{Z}_2^\mu \to \mathbb{Z}_2$ nazveme prostým součinem, pokud je tvaru

$$f(x_1, \ldots, x_\mu) = x_1^{i_1} \cdot \ldots \cdot x_\mu^{i_\mu} ,$$

kde $\cdot$ značí logický součin a $i_1, \ldots, i_\mu \in \{0, 1\}$. Jde tedy o logický součin některých z proměnných $x_1, \ldots, x_\mu$, z nichž žádná není negována – odtud prostý. Počet exponentů $i_1, \ldots, i_\mu$ rovných 1 označujeme jako řád prostého součinu.

Například pro $\mu = 3$ existují tyto prosté součiny:

Řád	Prosté součiny
0	1
1	$x_1, \quad x_2, \quad x_3$
2	$x_1 \cdot x_2, \quad x_1 \cdot x_3, \quad x_2 \cdot x_3$
3	$x_1 \cdot x_2 \cdot x_3$

Každou logickou funkci μ proměnných lze plně určit tabulkou jejich výstupů pro všech 2^μ možných různých ohodnocení vstupních proměnných – pravdivostní tabulkou. Totéž musí platit i pro prosté součiny. Binárním zápisem prostého součinu $f(x_1, \ldots, x_\mu)$ budeme rozumět 2^μ-tici

$$(f_0, f_1, \ldots, f_{2^m-1}) \in \mathbb{Z}_2^{2^\mu} ,$$

kde pro každé $j \in \mathbb{N}, j < 2^\mu$ s binárním rozvojem $j_\mu j_{\mu-1} \ldots j_1$ platí

$$f_j = f(j_1, \ldots, j_\mu) ,$$

čili nejméně významný bit j_1 čísla j udává ohodnocení vstupní proměnné x_1, druhý nejméně významný bit j_2 ohodnocení proměnné x_2, atd. Pak f_j udává hodnotu prostého součinu f při tomto ohodnocení. Jde tedy o pravdivostní tabulku zapsanou do řádku s pevně daným pořadím ohodnoceních vstupních proměnných[1].

Například pro prostý součin $f(x_1, x_2) = x_1 \cdot x_2$ je binární zápis ve tvaru

$$(f_0, f_1, f_2, f_3) \;=\; (f(0,0), f(1,0), f(0,1), f(1,1)) \;=\; (0,0,0,1) .$$

RM kód je dán dvojicí parametrů $\rho, \mu \in \mathbb{N}$ splňujících $1 < \rho \leq \mu$. Takový kód pak označujeme $RM(\rho, \mu)$ kód. Generující matice G kódu $RM(\rho, \mu)$ je tvořena několika podmaticemi

$$G = \begin{pmatrix} {}^0G \\ {}^1G \\ \vdots \\ {}^\rho G \end{pmatrix} ,$$

kde řádky podmatice ${}^\ell G$ jsou tvořeny binárními zápisy všech prostých součinů nad μ proměnnými řádu ℓ. Označíme-li $u_1 < \ldots < u_l$ indexy proměnných x zahrnutých do součinu, jsou prosté součiny v řádcích matice ${}^\ell G$ řazeny lexikograficky dle ℓ-tic $(u_1, \ldots, u_l)$.

Každý binární zápis prostého součinu nad μ proměnnými je délky 2^μ, zároveň nad μ proměnnými existuje $\binom{\mu}{\ell}$ různých prostých součinů řádu ℓ. Tudíž rozměry matice ${}^\ell G$ jsou $\binom{\mu}{\ell} \times 2^\mu$. Pro délku kódu n a jeho dimenzi k tak platí

$$n = 2^\mu \qquad \text{a} \qquad k = \sum_{\ell=0}^{\rho} \binom{\mu}{\ell} .$$

Kódová vzdálenost C_{dist} a počet opravitelných chyb t jsou dány vztahy

$$C_{dist} = 2^{\mu-\rho} \qquad a \qquad t = \left\lfloor \frac{C_{dist} - 1}{2} \right\rfloor = \left\lfloor \frac{2^{\mu-\rho} - 1}{2} \right\rfloor .$$

Tedy při fixním μ se s klesajícím ρ zvyšuje korekční schopnost kódu. Zároveň ale zůstává délka kódu stejná a klesá jeho dimenze, neboli roste redundance.

[1] Některé zdroje však mohou uvádět jiné pořadí a odpovídajícím způsobem modifikovaný postup opravy.

Implementace

Konstrukci RM kódu provádí funkce `RMCode` se dvěma pozičními parametry ρ a μ. Matice G je tvořena postupně po podmaticích.

Matice ${}^0G$ je vždy tvořena jedním řádkem samých jedniček, její generování je při znalosti délky kódu n velmi jednoduché. Matice ${}^1G$ má vždy μ řádků a ve sloupcích obsahuje binární rozvoje všech celých čísel od 0 do $2^\mu - 1$. Aby její řádky odpovídaly binárním zápisům daných prostých součinů, musí být nejméně významné bity těchto rozvojů v prvním řádku matice a nejvíce významné bity v tom posledním. Pro generování binárních rozvojů byla použita systémová `Tuples` umožňující vytvořit všechny μ-tice nad množinou $\{0, 1\}$.

Řádky matice ${}^1G$ obsahují binární zápisy prostých součinů x_1 až x_μ. Zároveň platí, že pokud je prostý součin součinem několika prostých součinů, je jeho binární zápis taktéž součinem jejich binárních zápisů provedeným po složkách. Pro konstrukci matice ${}^\ell G$ jsou proto nejprve vygenerovány všechny podmnožiny množiny $\{1, \ldots, \mu\}$ velikosti ℓ, což provádí systémová `Subsets`. Pomocí každé z podmnožin jsou následně indexovány řádky matice ${}^1G$ a vypočten jejich součin po složkách. Výsledky tvoří řádky matice ${}^\ell G$.

Například binární zápis prostého součinu $x_1 \cdot x_3$ patřícího do matice ${}^2G$, který lze chápat jako součin dvou prostých součinů x_1 a x_3, je možné získat jako součin po složkách 1. a 3. řádku matice ${}^1G$.

Příklad 1 *Konstrukce RM kódu s parametry $\rho = 2$ a $\mu = 4$.*

```
In[ ]:= RMCode[2, 4]
```

$\rho = 2$ → up to 2-tuples

$\mu = 4$ → variables: x_1, x_2, x_3, x_4

Generator matrix G:

$$
{}^0G \begin{pmatrix} 1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1 \end{pmatrix} \quad 1
$$

$$
{}^1G \begin{pmatrix} 0&1&0&1&0&1&0&1&0&1&0&1&0&1&0&1 \\ 0&0&1&1&0&0&1&1&0&0&1&1&0&0&1&1 \\ 0&0&0&0&1&1&1&1&0&0&0&0&1&1&1&1 \\ 0&0&0&0&0&0&0&0&1&1&1&1&1&1&1&1 \end{pmatrix} \quad \begin{matrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{matrix}
$$

$$
{}^2G \begin{pmatrix} 0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&1 \\ 0&0&0&0&0&1&0&1&0&0&0&0&0&1&0&1 \\ 0&0&0&0&0&0&0&0&0&1&0&1&0&1&0&1 \\ 0&0&0&0&0&0&1&1&0&0&0&0&0&0&1&1 \\ 0&0&0&0&0&0&0&0&0&0&0&1&1&0&0&1&1 \\ 0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1 \end{pmatrix} \quad \begin{matrix} x_1 \cdot x_2 \\ x_1 \cdot x_3 \\ x_1 \cdot x_4 \\ x_2 \cdot x_3 \\ x_2 \cdot x_4 \\ x_3 \cdot x_4 \end{matrix}
$$

Code parameters:

$n = 2^\mu = 2^4 = 16$

$k = \sum_{i=0}^{\rho} \binom{\mu}{i} = \binom{4}{0} + \binom{4}{1} + \binom{4}{2} = 1 + 4 + 6 = 11$

$t = \lfloor (2^{\mu-\rho} - 1)/2 \rfloor = \lfloor (4 - 1)/2 \rfloor = 1$

```
Out[ ]= ⟨|type → RMCode, n → 16, k → 11, GF → 2, ρ → 2, μ → 4, t → 1,
 G → {{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, {0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1},
   {0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1}, {0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1},
   {0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1}, {0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1},
   {0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1}, {0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1},
   {0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1}, {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1},
   {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1}}, EC → 1 error|⟩
```

Funkci `RMCode` *jsou předány parametry kódu* ρ *a* μ*. Jejím výstupem je kód reprezentovaný asociací, mezi jejíž klíče patří mimo jiné délka kódu* n *a jeho dimenze* k*, dále parametry* ρ *a* μ*, generující matice* G *a počet opravitelných chyb* t*. Dodatečný výpis rozepisuje matici* G *do podmatic* ${}^{\ell}G$ *a každý řádek je vpravo doplněn o prostý součin, jehož binární reprezentaci představuje. V závěru je rozepsán výpočet délky kódu, dimenze a počtu opravitelných chyb.*

4.2.2 Kódování

Kódování informačního slova $a \in \mathbb{Z}_2^k$ je prováděno standardně násobením generující maticí $G \in \mathbb{Z}_2^{k \times n}$. Kódové slovo $b \in \mathbb{Z}_2^n$ získáme jako

$$b = a \cdot G\,,$$

kde a a b jsou považovány za řádkové vektory a $\cdot$ značí maticové násobení.

Implementace

Balíček *RMCode* neobsahuje dedikovanou funkci pro kódování. Používá se funkce `MatrixEncode` z balíčku *CommonCode*.

Příklad 2 *Zakódování informačního slova generující maticí kódu* $RM(1,3)$*.*

```
In[ ]:= K = RMCode[1, 3, NoPrint → True];
        MatrixEncode[K["G"], {0, 1, 1, 0}]
Out[ ]= {0, 1, 1, 0, 0, 1, 1, 0}
```

Nejprve je vytvořen kód K *s parametry* $\rho = 1$ *a* $\mu = 3$ *a zakázaným dodatečným výstupem. Následně je informační slovo 0110 zakódováno pomocí generující matice* G*, která je jedním z klíčů v asociaci reprezentující kód* K*.*

4.2.3 Oprava a dekódování

Oprava a dekódování probíhají u RM kódů najednou postupem označovaným jako majoritní dekódování.

Vstupem je přijaté slovo $c \in \mathbb{Z}_2^n$, výstupem dekódované slovo $d \in \mathbb{Z}_2^k$. To je pro účely majoritního dekódování rozděleno do podslov ${}^{0}d$ až ${}^{\rho}d$ podobně jako matice G. Podslovo ${}^{\ell}d$ obsahuje přesně tolik bitů, kolik řádků má podmatice ${}^{\ell}G$. Dekódování probíhá postupně v krocích od ${}^{\rho}d$ až po ${}^{0}d$.

Uvažujme krok zpracovávající podslovo ${}^{\ell}d$. Pro každý z bitů ${}^{\ell}d$ je z generující matice vyvozeno $2^{\mu-\ell}$ nezávislých rovnic. Z nichž každá je tvořena součtem 2^{ℓ} bitů slova c. Je-li slovo c bez chyby, měly by všechny tyto rovnice dávat stejné řešení pro daný bit slova ${}^{\ell}d$. V případě, že však slovo c obsahuje chyby, mohou dávat řešení různá. V tom případě je vybrána taková hodnota daného bitu, při níž je splněna většina rovnic – odtud majoritní dekódování. Pokud c

neobsahuje více než t chyb, nemůže nastat nerozhodná situace a všechny bity slova ${}^{\ell}d$ jsou tímto způsobem úspěšně dekódovány. V závěru je od slova c odečten násobek ${}^{\ell}d \cdot {}^{\ell}G$. V dalším kroku je zpracováno podslovo ${}^{\ell-1}d$ s upravenou hodnotou c.

Při konstrukci rovnic pro jednotlivé bity slova d se používá množina značená $M(w)$, která pro dané $w \in \mathbb{N}_0$ obsahuje všechna $v \in \mathbb{N}_0$ taková, že $v \leq w$ a zároveň má v v binárním rozvoji jedničky jen tam, kde je má číslo w.

Například platí

$$M(2) = \{0, 2\} \qquad \text{nebo} \qquad M(5) = \{0, 1, 4, 5\}\,.$$

Sestavování rovnic pro bit d_u, kde $u \in \{1, \ldots, k\}$ s číslováním odpředu, závisí na prostém součinu v u-tém řádku matice G. Je-li tento součin zapsán ve tvaru

$$x_1^{i_1} \cdot \ldots \cdot x_\mu^{i_\mu} \qquad \text{pro} \qquad i_1, \ldots, i_\mu \in \{0, 1\}\,,$$

pak značíme i číslo z $\mathbb{N}_0$, které má binární rozvoj $i_\mu \ldots i_1$. Pro bit d_u existují rovnice

$$d_u = \sum_{x \in M(i)} c_{x+y+1} \qquad \text{pro každé} \qquad y \in M(n-i-1)\,,$$

kde n je délka kódu a jednotlivé bity slova c jsou indexovány odpředu od 1. Každá z rovnic je suma přes vybrané bity slova c, rovnic existuje tolik, kolik prvků má množina $M(n-i-1)$.

Implementace

Majoritní dekódování provádí funkce `RMCorrectAndDecode`, jejímiž dvěma parametry jsou kód reprezentovaný asociací a přijaté slovo c.

Pro každý bit d_u je nutné najít prostý součin na u-tém řádku matice G. Je-li systémové `Subsets` předána množina proměnných $\{x_1, \ldots, x_\mu\}$, vrátí její podmnožiny přesně v pořadí, v jakém jsou zahrnuty do prostých součinů v řádcích matice G. Proměnné figurující v prostém součinu na u-tém řádku G tak lze získat jako u-tou z podmnožin.

Dále musí funkce pro daný bit d_u a odpovídající prostý součin umět určit hodnotu i daného součinu. Například pro součin $x_1 \cdot x_3 \cdot x_4$ je i získáno (nezávisle na počtu proměnných μ) jako součet

$$i \quad = \quad 2^{1-1} + 2^{3-1} + 2^{4-1} \quad = \quad 1 + 4 + 8 \quad = \quad 13\,.$$

Hledání množin $M(w)$ je implementováno pomocí systémové `Select`, která ze všech čísel $\{0, \ldots, w\}$ vybere ta v, pro něž `BitOr[v,w] == w`, neboli výsledek operace OR po bitech těchto dvou čísel je roven w. Tedy v neobsahuje jedničky na jiných pozicích než w.

Řešení rovnic je pak jednoduše provedení operace XOR přes všechny bity slova c indexované množinami $M(i)$ a $M(n-i-1)$.

Ačkoliv Mathematica nabízí funkci `Majority`, byla pro určení řešení většiny rovnic použita systémová `Median` vracející, jak název napovídá, medián zadaných hodnot. Tímto způsobem je možné detekovat nerozhodné situace, kdy polovina rovnic udává hodnotu bitu 0 a druhá 1. Medián v takovém případě vyjde 0.5, zatímco `Majority` pracuje pouze nad logickými hodnotami `True` a `False` a při nerozhodném stavu vrací automaticky `False`. V případě detekování nerozhodného stavu je uživatel o této skutečnosti informován a dekódování je přerušeno s chybovou hláškou.

Příklad 3 *Majoritní dekódování slova kódu $RM(1,3)$ s chybou na druhém bitu.*

```
In[ ]:= K = RMCode[1, 3, NoPrint → True];
        RMCorrectAndDecode[K, {0, 0, 1, 0, 0, 1, 1, 0}]

        ρ = 1, μ = 3, n = 8, k = 4, t = 1

        ──────────────────────────────────────────────────────────────────────

        Majority decoding ¹d = {d₂, d₃, d₄}   → ¹d = {1, 1, 0}

        d₂ = c₁+c₂  = 0+0  = 0          d₃ = c₁+c₃  = 0+1  = 1          d₄ = c₁+c₅  = 0+0  = 0
           = c₃+c₄  = 1+0  = 1             = c₂+c₄  = 0+0  = 0             = c₂+c₆  = 0+1  = 1
           = c₅+c₆  = 0+1  = 1             = c₅+c₇  = 0+1  = 1             = c₃+c₇  = 1+1  = 0
           = c₇+c₈  = 1+0  = 1             = c₆+c₈  = 1+0  = 1             = c₄+c₈  = 0+0  = 0
                    d₂ = 1                          d₃ = 1                          d₄ = 0

        c = c - ¹d·¹G
        c = {0, 0, 1, 0, 0, 1, 1, 0} - {0, 1, 1, 0, 0, 1, 1, 0}
        c = {0, 1, 0, 0, 0, 0, 0, 0}

        ──────────────────────────────────────────────────────────────────────

        Majority decoding ⁰d = {d₁}   → ⁰d = {0}

        d₁ = c₁  = 0
           = c₂  = 1
           = c₃  = 0
           = c₄  = 0
           = c₅  = 0
           = c₆  = 0
           = c₇  = 0
           = c₈  = 0
            d₁ = 0

        c = c - ⁰d·⁰G
        c = {0, 1, 0, 0, 0, 0, 0, 0} - {0, 0, 0, 0, 0, 0, 0, 0}
        c = {0, 1, 0, 0, 0, 0, 0, 0}

        ──────────────────────────────────────────────────────────────────────

        Returning d = {⁰d, ¹d} = {d₁, d₂, d₃, d₄}

Out[ ]= {0, 1, 1, 0}
```

Nejprve je definován kód K s parametry $\rho = 1$ a $\mu = 3$ a se zakázaným dodatečným výpisem. Poté je pomocí tohoto kódu opraveno a dekódováno slovo $c = 00100110$ obsahující chybu na druhém bitu.

Výstupem funkce `RMCorrectAndDecode` *je dekódované slovo d. Dodatečný výpis nejprve uvádí základní informace o použitém kódu. Následně je rozepsána oprava po jednotlivých krocích oddělených horizontálními čarami. V každém kroku jsou pro všechny bity odpovídajícího ${}^{\ell}d$ vypsány všechny rovnice a jejich*

řešení. Majoritní hodnota je následně zobrazena v rámečku. V závěru každého kroku je rozepsána úprava c před vstupem do toho dalšího.

Je patrné, že všechny rovnice, v nichž je zahrnut chybový druhý bit c_2, vracejí opačný výsledek než zbylé rovnice.

Dodatečný výpis funkce `RMCorrectAndDecode` je příliš obsáhlý na to, aby zahrnoval ještě popis konstrukce jednotlivých rovnic. A to i u tak malého kódu jako je $RM(1,3)$. Z tohoto důvodu byla do balíčku *RMCode* přidána další funkce popisující tvorbu rovnic. Nese název `RMConstructEquations`. Bere jako parametr kód reprezentovaný asociací a index u bitu d_u, pro nějž mají být rovnice sestaveny.

Příklad 4 *Tvorba rovnic pro 3. bit slova d u kódu $RM(1,3)$.*

```
In[•]:= K = RMCode[1, 3, NoPrint → True];
        RMConstructEquations[K, 3]

        ρ = 1, μ = 3, n = 8, k = 4
            → 3 variables: x1, x2, x3

        d3 corresponds to the 3. row of the matrix G: x2   →   x3 | x2 | x1
                                                               0  | 1  | 0    →   i = 2
                                                                                 n - i - 1 = 5

        M(2) = {0, 2}
        1 0 | 2
        0 0 | 0
        1 0 | 2

        M(5) = {0, 1, 4, 5}
        1 0 1 | 5
        0 0 0 | 0
        0 0 1 | 1
        1 0 0 | 4
        1 0 1 | 5

        + | 0  2
        --+------
        0 | 0  2    →   d3 = c1 + c3
        1 | 1  3    →   d3 = c2 + c4
        4 | 4  6    →   d3 = c5 + c7
        5 | 5  7    →   d3 = c6 + c8
```

Opět je nejprve definován kód K s parametry $\rho = 1$ a $\mu = 3$ a se zakázaným dodatečným výpisem. Následně je předán funkci `RMConstructEquations` *spolu s indexem bitu slova d.*

Funkce nemá žádný výstup. V rámci dodatečného výpisu však vypisuje postup zjištění hodnoty i z prostého součinu na třetím řádku matice G. Dále je rozepsáno složení množin $M(i)$ a $M(n-i-1)$. Každý řádek tabulky obsahuje jeden prvek množiny, nejprve zapsaný binárně, následně decimálně. Bity binárního zápisu jsou podbarveny podle jedniček a nul v zápisu zadaného i, případně $n-i-1$. Prvky množiny totiž mohou mít jedničky jen tam, kde je má zadané číslo.

V závěru jsou na základě prvků těchto množin sestaveny rovnice. Tabulka obsahuje součty $y+x$ pro $y \in M(n-i-1)$ a $x \in M(i)$. Každý řádek tabulky popisuje jednu rovnici. Pro získání indexu slova c je každý součet $y+x$ zvýšen o 1.

4.3 Goppa kódy

Goppa kódy mohou být jak binární, tak nebinární kódy generované maticí. Zde je uvažována pouze speciální třída ireducibilních binárních Goppa kódů, které vedle zabezpečení dat před šumem nalézají uplatnění také v kryptografii. Související funkce jsou implementovány v balíčku *GoppaCode* a ukázky jejich použití se nachází v notebooku `GoppaKody.nb`.

Nejprve je v sekci 4.3.1 popsána konstrukce kódu, následně v 4.3.2 kódování a na závěr v 4.3.3 oprava a dekódování.

Informace čerpány převážně z [2], také z [10] a [11].

4.3.1 Konstrukce kódu

Ireducibilní binární Goppa kód je dán ireducibilním polynomem stupně alespoň 1 nad tělesem $GF(2^m)$ pro $m \in \mathbb{N}$, $m > 1$. Tento polynom označujeme jako tzv. Goppův polynom. Budeme jej značit $G(x)$, jeho stupeň t a těleso $GF(2^m)$, z něhož má $G(x)$ koeficienty, zkráceně jako $\mathbb{F}$.

Mějme Goppův polynom $G(x) \in \mathbb{F}[x]$ a nalezněme všechny[2] prvky $\mathbb{F}$, které nejsou jeho kořeny. Označme dále n počet takovýchto prvků. Uspořádáme-li je do n-tice, získáme takzvanou podporu daného kódu označovanou L. Dostáváme tedy

$$L = (L_1, \ldots, L_n) \in \mathbb{F}^n \qquad \text{kde} \qquad \forall i, j : L_i \neq L_j \wedge G(L_i) = 0\,.$$

Značení n není náhodou, tato hodnota skutečně udává délku výsledného kódu.

Při konstrukci kódu je nejprve sestavena kontrolní matice kódu H. Ta je pak klasickým způsobem převedena na generující matici G. Kontrolní matici H získáme součinem tří dalších matic K, V a D.

Označme koeficienty $G(x)$ jako $g_t, \ldots, g_0 \in \mathbb{F}$ tak, že

$$G(x) = g_t \cdot x^t + \ldots + g_1 \cdot x + g_0\,.$$

Dále nechť $G(L_i)^{-1} \in \mathbb{F}$ značí pro všechna $1 \leq i \leq n$ inverzi prvku $\mathbb{F}$ získaného dosazením L_i z podpory za x do $G(x)$. Z podmínky, že žádný z prvků L_i není kořenem $G(x)$, víme, že jeho dosazením do $G(x)$ nezískáme nulový prvek, a inverze tedy bude vždy existovat. Matice $K \in \mathbb{F}^{t\times t}$, $V \in \mathbb{F}^{t\times n}$ a $D \in \mathbb{F}^{n\times n}$ jsou tvaru

$$K = \begin{pmatrix} g_t & 0 & \ldots & 0 \\ g_{t-1} & g_t & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ g_1 & g_2 & \ldots & g_t \end{pmatrix}, \qquad V = \begin{pmatrix} 1 & 1 & \ldots & 1 \\ L_1 & L_2 & \ldots & L_n \\ \vdots & \vdots & \ddots & \vdots \\ L_1^{t-1} & L_2^{t-1} & \ldots & L_n^{t-1} \end{pmatrix},$$

[2] Zdroje [10] a [11] striktně nevyžadují, aby v L byly opravdu všechny prvky $\mathbb{F}$, jenž nejsou kořeny $G(x)$. Takové kódy zde nejsou uvažovány.

$$D = \begin{pmatrix} G(L_1)^{-1} & 0 & \dots & 0 \\ 0 & G(L_2)^{-1} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & G(L_n)^{-1} \end{pmatrix}.$$

Pro kontrolní matici platí $H = K \cdot V \cdot D$, tedy $H \in \mathbb{F}^{t \times n}$. Pro převod na binární matici využijeme faktu, že každý prvek $\mathbb{F} = GF(2^m)$ lze zapsat jako m-tici nad $\mathbb{Z}_2$. Matice H je tak upravena na matici ze $\mathbb{Z}_2^{mt \times n}$ rozepsáním každého prvku $\mathbb{F}$ na sloupec m hodnot ze $\mathbb{Z}_2$. Sloupec je získán transpozicí řádkového zápisu m-tice.

Zdroj [2] uvádí, že je možné matici K ze součinu vynechat. Výsledná matice bude splňovat tu vlastnost, že pro všechna kódová slova c je syndrom $s = c \cdot H^\top$ nulový. Oprava však na základě tohoto syndromu neproběhne úspěšně.

Polynom $G(x)$ neslouží jako generující polynom kódu. Kód je generován maticí G získanou standardním způsobem dle vztahu $G \cdot H^\top = O$, kde O značí nulovou matici.

Výše uvedený popis konstrukce by bylo možné použít i pro reducibilní Goppa kód daný reducibilním Goppovým polynomem. Omezením se na ireducibilní $G(x)$ však získáme možnost blíže určit některé vlastnosti výsledného kódu. Ireducibilní polynom nemá žádné kořeny, součástí podpory L jsou tudíž všechny prvky $\mathbb{F} = GF(2^m)$. Navíc je zaručeno, že matice H obsahuje pouze lineárně nezávislé řádky, a redundance kódu je tak rovna počtu jejích řádků mt. Pro délku kódu n a jeho dimenzi k dostáváme

$$n = 2^m \qquad \text{a} \qquad k \;=\; n - mt \;=\; 2^m - mt\,.$$

Zároveň ani značení stupně ireducibilního $G(x)$ symbolem t není náhodou, jelikož pomocí odpovídajícího Goppa kódu lze vždy opravit t chyb.

Implementace

Jelikož konstrukce ireducibilního Goppa kódu zcela závisí na volbě ireducibilního Goppa polynomu $G(x)$, byla vytvořena speciální funkce `GoppaPolynomial` umožňující tyto polynomy generovat.

Funkce `GoppaPolynomial` je ve skutečnosti pouze obálkou funkce pod názvem `GFIrreduciblePolynomial` z balíčku *CommonGFCode*, která slouží k hledání ireducibilních polynomů nad konečnými tělesy a je blíže popsána na straně 13. `GoppaPolynomial` přidává pouze kontrolu, zdali lze daný ireducibilní polynom použít pro konstrukci Goppa kódu. Stupeň t Goppa polynomu nad $GF(2^m)$ je totiž omezen vztahem $t < 2^m/m$, aby redundance kódu mt nepřevýšila jeho délku $n = 2^m$.

Funkce má dva poziční parametry, první specifikuje těleso, ze kterého mají být vybírány koeficienty $G(x)$. Druhý parametr udává jeho požadovaný stupeň t. Je-li jako specifikace tělesa zadán ireducibilní polynom nad $\mathbb{Z}_2$ stupně

m, je tento použit jako definující polynom tělesa $GF(2^m)$, z něhož jsou koeficienty vybírány. Je-li zadáno celé číslo $m > 1$, bude pro konstrukci $GF(2^m)$ použit výchozí ireducibilní polynom daného stupně.

Vedle pozičních parametrů má funkce `GoppaPolynomial` ještě 2 pojmenované parametry převzaté z funkce `GFIrreduciblePolynomial`. Parametr `Source` udává, zdali má být nejdříve polynom hledán v databázi ireducibilních polynomů, která je blíže popsána na straně 13, nebo má být rovnou volen náhodně. Náhodná volba polynomu je realizována výběrem t prvků $GF(2^m)$, které jsou považovány za jeho koeficienty. Náhodné polynomy jsou generovány tak dlouho, dokud test ireducibility nevrátí uspokojivý výsledek. Testování ireducibility provádí funkce `GFIrreduciblePolynomialQ` z *CommonGFCode* popsaná na straně 12. Pojmenovaný parametr `TimeLimit` umožňuje omezit dobu běhu funkce. Při jejím překročení je funkce ukončena s chybovým výpisem informujícím o počtu otestovaných polynomů. Není-li zadán žádný z pojmenovaných parametrů, jsou ireducibilní polynomy hledány s využitím databáze a dobou běhu limitovanou na 30 vteřin.

Příklad 1 *Ireducibilní Goppa polynom nad $GF(2^3)$ stupně 2.*

```
In[•]:= GoppaPolynomial[3, 2]

m = 3  → coefficients from GF(2^3) with default defining polynomial z^3 + z + 1
t = 2  → degree 2

Irreducible: true
Source: table
```

Notation for elements of GF(2^3)	G(x)
Polynomial in z	$(1)x^2 + (z)x + (z)$
Binary	$001\,x^2 + 010\,x + 010$
Decimal	$1x^2 + 2x + 2$
Mathematica	$\{0,0,1\}_{GF(8)}\,x^2 + \{0,1,0\}_{GF(8)}\,x + \{0,1,0\}_{GF(8)}$

```
Out[•]= {{0, 0, 1}_GF(8), {0, 1, 0}_GF(8), {0, 1, 0}_GF(8)}
```

Funkci je předána specifikace tělesa jako celé číslo 3, takže je použit výchozí polynom pro definici $GF(2^3)$. Druhý parametr je požadovaný stupeň Goppa polynomu 2.

Výstupem funkce je polynom jako seznam koeficientů z $GF(2^3)$. Tato forma je upřednostněna před polynomem v proměnné x, neboť Mathematica pak neumí jednotlivé monomy správně seřadit. Dodatečný výpis nejprve vysvětluje význam zadaných pozičních a pojmenovaných parametrů. I když v tomto případě nejsou žádné pojmenované parametry uvedeny, byly použity jejich výchozí hodnoty. Ireducibilní polynom je tedy nejprve hledán v databázi a délka běhu je omezena na 30 vteřin. Zde vidíme, že polynom v databázi nalezen byl, jak dokládá položka `Source`*. Limitace výstupu na seznam koeficientů je kompenzována tabulkou, ve které je výsledný Goppa polynom zapsán v hned několika notacích jako polynom v x.*

Jediným povinným vstupem funkce `GoppaCode` konstruující kód je ireducibilní Goppa polynom $G(x)$ zadaný jako seznam koeficientů nebo jako polynom v x. Volitelně lze jako druhý parametr předat funkci pro zobrazování prvků konečného tělesa v dodatečném výpisu. Ireducibilita polynomu není z důvodu časové náročnosti kontrolována, pouze v případě nalezení kořene je běh funkce ukončen s chybovou hláškou.

Ze zadaného polynomu je zjištěno těleso $\mathbb{F} = GF(2^m)$, z nějž jsou jeho koeficienty. Dále jsou nalezeny všechny prvky tohoto tělesa a z nich následně vybrána podpora kódu L. Pořadí prvků je vždy dle jejich decimálního zápisu. Zdroj [2] uvádí, že pro použití v kryptografii by mělo být pořadí voleno náhodně. Zde je však dána přednost faktu, že pro stejný $G(x)$ bude vždy vrácen stejný kód. Ověření, zda daný prvek $\mathbb{F}$ je kořenem $G(x)$, se provádí dosazením prvku za x do $G(x)$ systémovou `ReplaceAll` a porovnáním výsledku s nulovým prvkem $\mathbb{F}$, což za použití balíčku *FiniteFields* pro reprezentaci prvků $\mathbb{F}$ znamená jednoduše porovnání s nulou. Invertování prvku $\mathbb{F}$ reprezentovaného pomocí balíčku *FiniteFields* je taktéž jednoduché, stačí jej umocnit na -1. Toho je využito při sestavování matice D.

Násobení matic K, V a D nad $\mathbb{F}$ je provedeno standardně systémovou `Dot`. Při převodu prvků matice H z $\mathbb{F}$ na sloupec binárních hodnot je využito faktu, že zápis seznamem koeficientů ze $\mathbb{Z}_2$ je součástí reprezentace každého prvku $\mathbb{F}$ pomocí *FiniteFields*.

Převod binární kontrolní matice H na generující matici G je proveden pomocí funkce `ConvertToG` z *CommonCode*, která interně využívá systémovou `NullSpace`. Převod se v případě, že H vyjde regulární, nemusí povést. Znamená to totiž, že redundance kódu je rovna jeho délce a nezbývá žádný prostor pro informační bity. Tato skutečnost je detekována a nastane-li, je uživatel upozorněn chybovou hláškou.

Příklad 2 *Ireducibilní Goppa kód daný polynomem nad $GF(2^3)$ stupně 2.*

Nejprve je pomocí `GoppaPolynomial` *se zakázaným dodatečným výpisem získán ireducibilní Goppa polynom. Ten je následné předán funkci* `GoppaCode` *spolu s funkcí* `GFToBinaryString` *z CommonGFCode, která způsobí zobrazování prvků konečného tělesa v dodatečném výpisu jako řetězec binárních hodnot.*

Výstupem funkce je kód reprezentovaný asociací, mezi jejíž klíče patří mimo jiné délka kódu n, dimenze kódu k, Goppa polynom $G(x)$, podpora L, kontrolní matice H a generující matice G.

Dodatečný výpis tvoří nejprve polynom $G(x)$ a z něj zjištěné parametry m a t, z kterých je následně odvozena délky kódu n, jeho dimenze k a korekční schopnost. Následně jsou v tabulce rozepsány všechny prvky L_i podpory spolu s hodnotou $G(L_i)^{-1}$, která se používá při konstrukci matice D. Poté je rozepsána konstrukce matic K, V, D a výsledné H. Drobným nešvarem je to, že nulové prvky $GF(2^3)$ jsou zobrazeny jako jediná nula. To je způsobeno tím, že při použití systémového balíčku FiniteFields pro práci s tělesy jsou nulové prvky

reprezentovány nulou typu `Integer` *a zobrazovací funkce* `GFToBinaryString` *nahrazuje pouze výrazy s hlavičkou* `GF`.

Při použití binárního zápisu pro prvky $GF(2^3)$ je dobře viditelný způsob převodu matice H na binární. Názornost je ještě podpořena rozdělením binárního zápisu H na podmatice, z nichž každá odpovídá jednomu řádku původní matice nad $GF(2^3)$. Na závěr je vypsána generující matice G.

```
In[ ]:= goppaPolynomial = GoppaPolynomial[3, 2, NoPrint → True];
        GoppaCode[goppaPolynomial, GFToBinaryString]
```

$G(x) = 001\,x^2 + 010\,x + 010$

$m = 3 \quad \to \quad n = 2^m = 8, \quad k = n - m \cdot t = 2$

$t = 2 \quad \to \quad$ correcting 2 errors

Support: $L = (L_1, \ldots, L_n) \quad \forall i,\ L_i \in GF(2^3) \wedge G(L_i) \neq 0$

i	L_i	$G(L_i)^{-1}$
1	0	101
2	001	001
3	010	101
4	011	001
5	100	100
6	101	111
7	110	100
8	111	100

$$K = \begin{pmatrix} g_2 & 0 \\ g_1 & g_2 \end{pmatrix} = \begin{pmatrix} 001 & 0 \\ 010 & 001 \end{pmatrix}$$

$$V = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ L_1 & L_2 & L_3 & L_4 & L_5 & L_6 & L_7 & L_8 \end{pmatrix} = \begin{pmatrix} 001 & 001 & 001 & 001 & 001 & 001 & 001 & 001 \\ 0 & 001 & 010 & 011 & 100 & 101 & 110 & 111 \end{pmatrix}$$

$$D = \begin{pmatrix} G(L_1)^{-1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & G(L_2)^{-1} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & G(L_3)^{-1} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & G(L_4)^{-1} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & G(L_5)^{-1} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & G(L_6)^{-1} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & G(L_7)^{-1} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & G(L_8)^{-1} \end{pmatrix} = \begin{pmatrix} 101 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 001 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 101 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 001 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 100 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 111 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 100 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 111 \end{pmatrix}$$

$$H = K \cdot V \cdot D = \begin{pmatrix} 101 & 001 & 101 & 001 & 100 & 111 & 100 & 111 \\ 001 & 011 & 0 & 001 & 101 & 011 & 110 & 110 \end{pmatrix}$$

$$H = \left(\begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \end{pmatrix} \right)$$

$$\left(\begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \end{pmatrix} \right)$$

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

```
Out[ ]= ⟨| type → GoppaCode, n → 8, k → 2, GF → 2, m → 3, t → 2, G(x) → {0, 1, 0}_GF(8) + x {0, 1, 0}_GF(8) + x^2 {0, 0, 1}_GF(8),
          L → {0, {0, 0, 1}_GF(8), {0, 1, 0}_GF(8), {0, 1, 1}_GF(8), {1, 0, 0}_GF(8), {1, 0, 1}_GF(8), {1, 1, 0}_GF(8), {1, 1, 1}_GF(8)},
          H → {{1, 0, 1, 0, 1, 1, 1, 1}, {0, 0, 0, 0, 0, 1, 0, 1}, {1, 1, 1, 1, 0, 1, 0, 1},
            {0, 0, 0, 0, 1, 0, 1, 1}, {0, 1, 0, 0, 0, 1, 1, 1}, {1, 1, 0, 1, 1, 1, 0, 0}},
          G → {{1, 0, 0, 1, 1, 1, 0, 1}, {0, 1, 1, 0, 0, 1, 1, 1}}, EC → 2 errors |⟩
```

4.3.2 Kódování

Kódování informačního slova $a \in \mathbb{Z}_2^k$ je prováděno standardně násobením generující maticí $G \in \mathbb{Z}_2^{k \times n}$. Kódové slovo $b \in \mathbb{Z}_2^n$ získáme jako

$$b = a \cdot G\,,$$

kde a a b jsou považovány za řádkové vektory a $\cdot$ značí maticové násobení.

Implementace

Balíček *GoppaCode* neobsahuje dedikovanou funkci pro kódování. Používá se funkce `MatrixEncode` z balíčku *CommonCode*.

Příklad 3 *Zakódování informačního slova generující maticí Goppa kódu daného polynomem nad $GF(2^3)$ stupně 2.*

```
In[ ]:= goppaPolynomial = GoppaPolynomial[3, 2, NoPrint → True];
        K = GoppaCode[goppaPolynomial, NoPrint → True];
        MatrixEncode[K["G"], {0, 1}]
Out[ ]= {0, 1, 1, 0, 0, 1, 1, 1}
```

Nejprve je získán ireducibilní Goppa polynom funkcí `GoppaPolynomial` *se zakázaným dodatečným výpisem. Následně je tento předán funkci* `GoppaCode`*, která vrátí odpovídající Goppa kód reprezentovaný asociací. U této funkce je rovněž zakázán dodatečný výpis. Následně je informační slovo 01 zakódováno pomocí generující matice G, která je jedním z klíčů v asociaci.*

4.3.3 Oprava a dekódování

Pro opravu byl použit Pattersonův algoritmus tak, jak je popsaný v [2] a [10]. Oba zdroje zároveň uvádí, že tato metoda opravy není jediná.

Mějme kód délky n daný ireducibilním Goppa polynomem $G(x)$ stupně t nad tělesem $\mathbb{F} = GF(2^m)$. Dále přijaté slovo $c \in \mathbb{Z}_2^n$. Postup jeho opravy je následující:

1. Pomocí kontrolní matice H vypočteme syndrom $s = c \cdot H^\top \in \mathbb{Z}_2^{mt}$. Je-li nulový, c je kódovým slovem, a nevyžaduje tak opravu. Tento syndrom lze rovněž interpretovat jako polynom $s(x) \in \mathbb{F}[x]$ tak, že s rozdělíme na t m-tic, z nichž každá odpovídá jednomu prvku $\mathbb{F}$. A prvky $\mathbb{F}$ pak tvoří seznam koeficientů polynomu $s(x)$ s nejvyšším členem na první pozici.

2. Vypočteme $r(x) = \sqrt{x - s(x)^{-1}}$ v okruhu zbytkových tříd modulo $G(x)$. Neboli nalezneme takové $S(x) \in \mathbb{F}[x]$ stupně $< t$, že

$$s(x) \cdot S(x) \equiv 1 \pmod{G(x)}.$$

   A dále nalezneme takové $r(x) \in \mathbb{F}[x]$ stupně $< t$, že

$$r(x)^2 \equiv x - S(x) \pmod{G(x)}.$$

3. Nalezneme $\alpha(x) \in \mathbb{F}[x]$ stupně $\leq \lfloor \frac{t}{2} \rfloor$ a $\beta(x) \in \mathbb{F}[x]$ stupně $\leq \lfloor \frac{t-1}{2} \rfloor$ takové, že

$$\alpha(x) \equiv \beta(x) \cdot r(x) \pmod{G(x)}.$$

4. Vypočteme lokátor $\sigma(x) \in \mathbb{F}[x]$ stupně $\leq t$ dle předpisu

$$\sigma(x) = \alpha(x)^2 + x \cdot \beta(x)^2 \,.$$

5. Opravíme c invertováním i-tého bitu, pokud prvek podpory $L_i \in \mathbb{F}$ je kořenem $\sigma(x)$.

Pro ireducibilní $G(x)$ je existence inverze v 2. bodě zaručena, neboť zbytkové třídy modulo $G(x)$ tvoří těleso. Jediným prvkem bez inverze je pak nulový polynom. Ten však značí slovo bez chyby a jeho zpracování je ukončeno již v 1. kroku. Faktu, že zbytkové třídy modulo $G(x)$ tvoří těleso, konkrétně rozšířené těleso $GF((2^m)^t)$, je využito i při hledání odmocniny v tomtéž kroku. Odmocninou nulového prvku je nulový prvek. Ostatní prvky jsou součástí multiplikativní grupy tohoto tělesa, která je řádu $2^{mt} - 1$. Proto pro každý nenulový prvek $u \in GF((2^m)^t)$ platí

$$u^{2^{mt}-1} = 1 \qquad \text{a tedy} \qquad u^{2^{mt}} = u \qquad \text{a} \qquad \left(u^{\frac{2^{mt}}{2}}\right)^2 = u \,.$$

Neboli prvek $u^{\frac{2^{mt}}{2}} = u^{2^{mt-1}}$ je odmocninou prvku u. Odmocnina libovolného $u \in GF((2^m)^t)$ tedy existuje a je dána jednoznačně.

Vztah ze 4. bodu lze také zapsat jako

$$\alpha(x) = \beta(x) \cdot r(x) + \gamma(x) \cdot G(x) \,,$$

pro nějaké $\gamma(x) \in \mathbb{F}[x]$. A přesně taková $\alpha(x)$, $\beta(x)$ a $\gamma(x)$ lze hledat rozšířeným Euklidovým algoritmem se vstupními polynomy $G(x)$ a $r(x)$. Pouze je pro splnění požadavků třeba algoritmus zastavit ve chvíli, kdy stupeň $\alpha(x)$ klesne na nejvýše $\lfloor \frac{t}{2} \rfloor$ a stupeň $\beta(x)$ ještě nepřeroste $\lfloor \frac{t-1}{2} \rfloor$. Kde t značí stupeň $G(x)$.

Opravené slovo $c' \in \mathbb{Z}_2^n$ dekódujeme stejně jako u ostatních kódů generovaných maticí řešením soustavy

$$d \cdot G = c'$$

pro neznámý řádkový vektor $d \in \mathbb{Z}_2^k$ a danou generující matici kódu $G \in \mathbb{Z}_2^{k \times n}$. Je-li G v systematickém tvaru, lze dekódování provést jednoduchým vybrání prvních k bitů slova c'.

Implementace

Opravu provádí funkce `GoppaCorrect` se dvěma povinnými pozičními parametry – kódem reprezentovaným asociací a přijatým slovem c. Volitelně lze jako třetí argument předat funkci pro zobrazování prvků konečného tělesa v dodatečném výpisu.

Výpočet syndromu $s \in \mathbb{Z}_2^{mt}$ je proveden pomocí `MatrixSyndrome` z balíčku *CommonCode*. Ten je následně systémovou `Partition` rozdělen na m-tice.

Převod na prvky $\mathbb{F} = GF(2^m)$ je pak při reprezentaci konečných těles s využitím balíčku *FiniteFields* jednoduchý. Je však třeba brát v potaz opačnou konvenci zapisování polynomů seznamem koeficientů od nejnižšího členu po nejvyšší.

Pro hledání inverze modulo $G(x)$ musela být implementována vlastní verze rozšířeného Euklidova algoritmu, jelikož systémová `PolynomialExtendedGCD` nepracuje správně s polynomy nad $GF(2^m)$. K tomu byla využita funkce `PolyDivide` z balíčku *PolynomialCode*, která umí vypočíst podíl dvou polynomů jak nad $\mathbb{Z}_2$, tak nad $GF(2^m)$. Druhá speciální verze pak byla implementována pro hledání $\alpha(x)$ a $\beta(x)$ ve čtvrtém kroku opravy, kdy je třeba algoritmus zastavit ve chvíli, kdy stupeň $\alpha(x)$ klesne na $\leq \lfloor \frac{t}{2} \rfloor$.

Odmocnina $\sqrt{x - s(x)^{-1}}$ je hledána na základě popisu uvedeného výše umocněním polynomu $x - s(x)^{-1}$ jakožto prvku $GF((2^m)^t)$ na 2^{mt-1}. To je provedeno opakovaným mocněním na 2 s redukcí modulo $G(x)$ po každém z $mt - 1$ umocnění. Funkce `PolyMod` z *PolynomialCode* byla využita pro provádění redukce a systémová `Nest` pro její opakovanou aplikaci.

Kořeny lokátoru $\sigma(x) \in \mathbb{F}[x]$ jsou hledány hrubou silou přes všechny prvky podpory L, která je součástí asociace reprezentující kód.

Příklad 4 *Oprava slova s chybou na 2. a předposledním bitu pomocí ireducibilního Goppa kódu daného polynomem nad $GF(2^3)$ stupně 2.*

Nejprve je vytvořen Goppa polynom $G(x)$ a na jeho základě kód K. Funkci `GoppaCorrect` *je předán kód reprezentovaný asociací, přijaté slovo c a funkce* `GFToBinaryString` *mající za následek zobrazení prvků $GF(2^3)$ binárními řetězci.*

Výstupem funkce je opravené slovo c'. Dodatečný výpis nejprve zobrazuje polynom $G(x)$ a odpovídající hodnoty parametrů m a t. Následně je rozepsán výpočet syndromu a jeho převod z binárního slova na polynom nad $GF(2^3)$. Poté je ze syndromu dopočten polynom $r(x)$, který je dále spolu s $G(x)$ vstupem rozšířeného Euklidova algoritmu při hledání polynomů $\alpha(x)$ a $\beta(x)$. Jednotlivé kroky algoritmu jsou rozepsány do tabulky s tím, že v prvním řádků je umístěn polynom $G(x)$ a v druhém $r(x)$, jelikož je vždy nižšího stupně než $G(x)$. Výsledné hodnoty $\alpha(x)$, $\beta(x)$ jsou určeny posledním řádkem tabulky. V tomto případě tedy $\alpha(x) = r(x)$ a $\beta(x) = 1$. Z těchto polynomů je dále určen lokátor $\sigma(x)$. Tabulka v dolní části zobrazuje všechny prvky podpory L a hodnoty získané jejich dosazením do $\sigma(x)$. Nulové hodnoty indikují pozice chyb použité k sestavení chybového slova e, které je v závěru přičteno k přijatému slovu c, čímž je oprava dokončena. Chyby na 2. a předposledním bitu byly správně opraveny.

```
In[ ]:= K = GoppaCode[GoppaPolynomial[3, 2, NoPrint → True], NoPrint → True];
        GoppaCorrect[K, {0, 0, 1, 0, 0, 1, 0, 1}, GFToBinaryString]

        G(x) = 001 x^2 + 010 x + 010
            m = 3
            t = 2

        s(x) = c·H^T = {1, 0, 1, 1, 0, 1}
          → s(x) = 101 x + 101

        r(x) = √(x − s(x)^-1) = √(001x − (010 x + 110)) = √(011 x + 110)
          → r(x) = 010 x + 011

        Finding α(x) and β(x) such that α(x) ≡ β(x)·r(x) (mod G(x)) using EEA
          until deg(α(x)) ≤ ⌊t/2⌋ = 1 ∧ deg(β(x)) ≤ ⌊(t−1)/2⌋ = 0
```

i	$\alpha_i(x)$	$\gamma_i(x)$	$\beta_i(x)$
1	001 x^2 + 010 x + 010 (= G(x))	001	0
2	010 x + 011 (= r(x))	0	001

```
        σ(x) = α(x)^2 + x·β(x)^2 = (010 x + 011)^2 + 001 x·(001)^2
          → σ(x) = 100 x^2 + 001 x + 101

        Finding elemets of support L that are roots of σ to reconstruct the error
```

i	1	2	3	4	5	6	7	8
L_i	0	001	010	011	100	101	110	111
$\sigma(L_i)$	101	0	001	100	100	001	0	101
e_i	0	1	0	0	0	0	1	0

```
        c = {0, 0, 1, 0, 0, 1, 0, 1}
        e = {0, 1, 0, 0, 0, 0, 1, 0}
        c'= {0, 1, 1, 0, 0, 1, 1, 1}
Out[ ]= {0, 1, 1, 0, 0, 1, 1, 1}
```

Pro dekódování nebyla v balíčku *GoppaCode* vytvořena speciální funkce. Používá se `MatrixDecode` z balíčku *CommonCode*, která interně používá systémovou `LinearSolve` pro řešení soustavy lineárních rovnic.

Příklad 5 *Dekódování opraveného slova ireducibilního Goppa kódu daného polynomem stupně 2 nad tělesem* $GF(2^3)$.

```
In[ ]:= K = GoppaCode[GoppaPolynomial[3, 2, NoPrint → True], NoPrint → True];
        MatrixDecode[K["G"], {0, 1, 1, 0, 0, 1, 1, 1}]
Out[ ]= {0, 1}
```

Nejprve je vytvořen Goppa polynom a na jeho základě kód K. *Funkci* `MatrixDecode` *je předána generující matice kódu, která je jednou z hodnot v asociaci reprezentující kód, a opravené slovo* $c' = 01100111$. *Výstupem je dekódované slovo* $d = 01$.

4.4 Binární BCH kódy

Bose–Chaudhuri–Hocquenghem (zkráceně BCH) kódy jsou kódy generované polynomem pro opravu volitelného množství chyb. Tato podkapitola je věnována binárním BCH kódům, nebinární BCH kódy jsou popsány v podkapitole 4.7. Související funkce jsou implementovány v balíčku *BCHCode* a ukázky jejich použití se nachází v notebooku `BCHBinarniKody.nb`.

Nejprve je v sekci 4.4.1 popsána konstrukce kódu, následně v 4.4.2 kódování a na závěr v 4.4.3 oprava a dekódování.

Informace čerpány z [12], [13] a [7, s. 141–160].

4.4.1 Konstrukce kódu

Binární BCH kód je dán konečným tělesem $GF(2^m)$ pro $m \in \mathbb{N}, m > 1$ a parametrem $\delta \in \mathbb{N}$ udávajícím zaručenou kódovou vzdálenost. Výsledkem je kód délky $n = 2^m - 1$ s kódovou vzdáleností $C_{dist} \geq \delta$. Nevíme tedy sice, jaká přesně bude kódová vzdálenost výsledného kódu, a tedy i jeho korekční schopnost, ale je jisté, že nebude nižší než δ, a kód tak umožní opravu nejméně $\lfloor(\delta - 1)/2\rfloor$ chyb.

Ačkoliv je kód binární, k jeho konstrukci se využívá prvků $GF(2^m)$ a jejich minimálních polynomů. Minimální polynom prvku $\beta \in GF(2^m)$ je polynom $M_\beta(x)$ nad $\mathbb{Z}_2$ nejnižšího stupně takový, že β je jeho kořenem. Pro více informací ohledně jejich hledání, viz stranu 16.

Mějme nějaký primitivní prvek α tělesa $GF(2^m)$, tedy takový prvek α, že libovolný nenulový prvek $GF(2^m)$ lze zapsat jako α^i pro nějaké $i \in \mathbb{N}_0$. Označme dále $M_{\#i}(x)$ minimální polynom prvku α^i. Pak pro generující polynom kódu $G(x) \in \mathbb{Z}_2[x]$ platí

$$G(x) = LCM(M_{\#1}(x), M_{\#2}(x), \ldots, M_{\#\delta-2}(x), M_{\#\delta-1}(x))\,,$$

kde LCM značí nejmenší společný násobek daných minimálních polynomů. Polynom $G(x)$ je tedy polynomem nejnižšího řádu takovým, že všechny prvky $\alpha^1, \alpha^2, \ldots, \alpha^{\delta-2}, \alpha^{\delta-1}$ jsou jeho kořeny. Jeho stupeň r udává redundanci kódu, a tedy i jeho dimenzi $k = n - r$.

Kontrolní matice H nad $GF(2^m)$ je tvaru

$$H = \begin{pmatrix} (\alpha^1)^{n-1} & (\alpha^1)^{n-2} & \ldots & (\alpha^1)^2 & \alpha^1 & 1 \\ (\alpha^2)^{n-1} & (\alpha^2)^{n-2} & \ldots & (\alpha^2)^2 & \alpha^2 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ (\alpha^{\delta-1})^{n-1} & (\alpha^{\delta-1})^{n-2} & \ldots & (\alpha^{\delta-1})^2 & \alpha^{\delta-1} & 1 \end{pmatrix}.$$

V i-tém řádku obsahuje mocniny prvku α^i. V posledním sloupci se nachází nejnižší nulté mocniny rovné jednotkovému prvku $GF(2^m)$, v prvním sloupci pak nejvyšší mocniny, prvky umocněné na $n - 1$. Celkem tedy n sloupců. Binární podobu matice H získáme převodem každého prvku $GF(2^m)$ na sloupec

m binárních hodnot. Konkrétně transpozicí řádkového zápisu prvku $GF(2^m)$ pomocí m-tice koeficientů ze $\mathbb{Z}_2$. Výsledkem je matice rozměrů $m(\delta-1) \times n$ nad $\mathbb{Z}_2$. Může však obsahovat lineárně závislé řádky.

Z konstrukce minimálních polynomů plyne, že pro daný prvek $\beta = \alpha^i$ s minimálním polynomem $M_\beta(x)$ a minimální polynom $M_{\beta^2}(x)$ jeho druhé mocniny $\beta^2 = \alpha^{2i}$ platí

$$M_{\beta^2}(x) = M_\beta(x) \qquad \text{neboli} \qquad M_{\#2i}(x) = M_{\#i}(x)\,.$$

Z tohoto důvodu je možné ze vzorce pro $G(x)$ vynechat všechna $M_{\#j}$ pro j sudá.

Považujeme-li přijaté slovo $c \in \mathbb{Z}_2^n$ za koeficienty polynomu $C(x)$, existují dvě metody stanovení jeho syndromu. První způsob je z matice H standardním výpočtem $s = c \cdot H^\top$, kde $s \in \mathbb{Z}_2^{m(\delta-1)}$ lze interpretovat jako $(\delta-1)$ dílčích syndromů $s_1, \ldots, s_{\delta-1} \in GF(2^m)$ tak, že každá m-tice bitů s popisuje jeden prvek $GF(2^m)$. Druhým způsobem je výpočet z polynomu $C(x)$, kde pro každý dílčí syndrom s_i platí $s_i = C(\alpha^i)$ s tím, že

$$s_{2i} = C(\alpha^{2i}) = C((\alpha^i)^2) = C(\alpha^i)^2 = s_i^2\,.$$

Neboli každý dílčí syndrom se sudým indexem lze dopočíst umocněním syndromu s poloviční indexem. Některé metody opravy tak nemusí nutně používat všechny dílčí syndromy. Navíc pokud $s_i = 0$, pak i $s_{2i} = 0$. Pro slova bez chyb, tak sudé dílčí syndromy nepřinášejí žádnou další informaci.

V závislosti na metodě opravy a metodě výpočtu syndromů, může matice H nabývat různých podob. Pokud metoda opravy využívá jen liché dílčí syndromy a ty jsou počítány z matice H, je možné vypustit z H všechny podmatice o m řádcích odpovídající sudým mocninám primitivního prvku α. Výsledná matice bude stále kontrolní maticí daného kódu. Jsou-li dílčí syndromy počítány namísto z matice H z polynomu $C(x)$, je možné z H vypustit dokonce všechny lineárně závislé řádky bez ohledu na podmatice.

Implementace

Konstrukci BCH kódů má na starosti funkce `BCHCode`. Má dva povinné poziční parametry, první specifikuje těleso $GF(2^m)$, pomocí nějž má být kód konstruován. Druhý parametr udává požadovanou zaručenou kódovou vzdálenost δ. Je-li jako specifikace tělesa zadán ireducibilní polynom nad $\mathbb{Z}_2$ stupně m, je tento použit jako definující polynom tělesa $GF(2^m)$. Je-li zadáno celé číslo $m > 1$, bude pro konstrukci $GF(2^m)$ použit výchozí ireducibilní polynom daného stupně.

Pro reprezentaci prvků konečných těles je jako ve zbytku práce využit systémový balíček *FiniteFields*. Vlastní funkce pro práci s těmito prvky jsou implementovány v balíčku *CommonGFCode*. Například funkce sloužící k hledání minimálních polynomů `GFMinimalPolynomial`. Nebo funkce začínající

na `GFTo*` sloužící pro zobrazování prvků těles v různých formách zápisu. Některou z těchto funkcí je možné předat `BCHCode` jako třetí poziční parametr, a ovlivnit tak způsob zobrazování prvků v dodatečném výpisu.

Primitivní prvek α tělesa $GF(2^m)$ je nalezen pomocí `FieldExp` z *FiniteFields*, která pro zadané u vrátí u-tou mocninu primitivního prvku. Pro $u = 1$ tedy primitivní prvek samotný. Použití této funkce je však mírně komplikované, neboť je nejprve voláním `PowerListQ` z téhož balíčku nutné nastavit, že má být pro operace nad daným tělesem používána tabulka mocnin primitivního prvku. Tato metoda například umožňuje provádět rychleji operaci násobení, cenou za to je množství paměti nutné pro uložení této tabulky a čas vynaložený na její konstrukci. [14]

Hledání minimálních polynomů je, jak již bylo řečeno, prováděno pomocí `GFMinimalPolynomial` z balíčku *CommonGFCode*. Z konstrukce minimálních polynomů plyne, že minimální polynomy $M_\beta(x)$ a $M_\gamma(x)$ nějakých dvou prvků $\beta, \gamma \in GF(2^m)$ jsou soudělné jen tehdy, pokud $M_\beta(x) = M_\gamma(x)$. Nejmenší společný násobek minimálních polynomů $M_{\#1}(x), \ldots, M_{\#\delta-1}(x)$ při konstrukci $G(x)$ je tak možné nahradit jejich součinem po odebrání duplicitních výskytů všech polynomů. K tomu je využita systémová `DeleteDuplicates`.

Pro binární BCH kódy se zaručenou kódovou vzdáleností $\delta \leq 8$ umožňující opravu nejvýše 3 chyb existuje, a je používána, jednodušší metoda opravy než pro zbylé kódy. Tato metoda využívá pouze liché dílčí syndromy. V závislosti na velikosti δ to mohou být s_1, s_3 a s_5. Do kontrolní matice H jsou u těchto kódů v souladu s [12] zahrnuty jen řádky odpovídající lichým mocninám primitivního prvku α. Stejně tak mezi minimální polynomy při výpočtu generujícího polynomu $G(x)$ jsou zahrnuty pouze minimální polynomy lichých mocnin α. Metoda opravy 4 a více chyb u kódů s $\delta > 8$ využívá i sudých dílčích syndromů, při konstrukci $G(x)$ a H proto u těchto kódů není žádná z mocnin vynechána.

Příklad 1 *Konstrukce binárního BCH kódu se zaručenou kódovou vzdáleností $\delta = 5$ s využitím tělesa $GF(2^4)$.*

Jako specifikace tělesa $GF(2^m)$ je funkci `BCHCode` *předáno celé číslo $m = 4$, jako definující polynom pro něj tudíž bude využit výchozí ireducibilní polynom stupně 4. Druhý parametr udává požadovanou hodnotu δ. Posledním parametrem je vynuceno zobrazení prvků $GF(2^4)$ jako binárních řetězců v dodatečném výpisu.*

Výstupem funkce je kód reprezentovaný asociací. Mezi její klíče patří například délka kódu n a jeho dimenze k, generující polynom $G(x)$, primitivní prvek α nebo kontrolní matice H.

Dodatečný výpis začíná rekapitulací zadaných hodnot. Následuje přehled mocnin primitivního prvku α spolu s jejich minimálními polynomy. V tomto případě je $\delta \leq 8$, takže sudé mocniny jsou vynechány. Následuje výpočet generujícího polynomu $G(x)$. Ten je nejprve rozepsán jako nejmenší společný násobek polynomů z tabulky, následně jako součin po odstranění duplicit. Na

základě stupně $G(x)$ je stanovena redundance a dimenze kódu k. Závěr výpisu je věnován konstrukci kontrolní matice H. Ta je nejprve rozepsána symbolicky jako mocniny primitivního prvku, následně jako matice nad $GF(2^4)$ a nakonec jako binární matice po rozepsání prvků $GF(2^4)$ do sloupců. Při zobrazení prvků tělesa binárními řetězci je dobře patrný způsob převodu. Názornost je dále umocněna rozdělením H do podmatic, z nichž každá odpovídá jednomu řádku matice H v předchozí formě zápisu.

```
In[ ]:= BCHCode[4, 5, GFToBinaryString]
```

GF(2^4), δ = 5, n = 15

Minimal polynomials:

i	α^i	$M_{\#1}(x)$
1	0010	$x^4 + x + 1$
3	1000	$x^4 + x^3 + x^2 + x + 1$

$G(x) = \text{LCM}[M_{\#1}(x), M_{\#3}(x)]$

$G(x) = M_{\#1}(x) \cdot M_{\#3}(x)$

$G(x) = (x^4 + x + 1) \cdot (x^4 + x^3 + x^2 + x + 1)$

$G(x) = x^8 + x^7 + x^6 + x^4 + 1$

$k = n - \deg[G(x)] = 15 - 8 = 7$

$$H = \begin{pmatrix} \alpha^{14} & \alpha^{13} & \alpha^{12} & \alpha^{11} & \alpha^{10} & \alpha^9 & \alpha^8 & \alpha^7 & \alpha^6 & \alpha^5 & \alpha^4 & \alpha^3 & \alpha^2 & \alpha & 1 \\ (\alpha^3)^{14} & (\alpha^3)^{13} & (\alpha^3)^{12} & (\alpha^3)^{11} & (\alpha^3)^{10} & (\alpha^3)^9 & (\alpha^3)^8 & (\alpha^3)^7 & (\alpha^3)^6 & (\alpha^3)^5 & (\alpha^3)^4 & (\alpha^3)^3 & (\alpha^3)^2 & \alpha^3 & 1 \end{pmatrix}$$

$$H = \begin{pmatrix} 1001 & 1101 & 1111 & 1110 & 0111 & 1010 & 0101 & 1011 & 1100 & 0110 & 0011 & 1000 & 0100 & 0010 & 0001 \\ 1111 & 1010 & 1100 & 1000 & 0001 & 1111 & 1010 & 1100 & 1000 & 0001 & 1111 & 1010 & 1100 & 1000 & 0001 \end{pmatrix}$$

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

```
Out[ ]= ⟨| type → BCHCode, n → 15, k → 7, GF → 2, δ → 5, G(x) → 1 + x^4 + x^6 + x^7 + x^8,
 α → {0, 0, 1, 0}_GF(16), H → {{1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0},
   {0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0}, {0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0},
   {1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1}, {1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0},
   {1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0}, {1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0},
   {1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1}}, EC → at least 2 errors |⟩
```

4.4.2 Kódování

Kódování u binárních BCH kódů je prováděno jako u ostatních kódů generovaných polynomem. A to tak, že je informační slovo $a \in \mathbb{Z}_2^k$ považováno za seznam koeficientů polynomu $A(x)$, který je vynásoben generujícím polynomem $G(x)$ stupně r. Kódové slovo $b \in \mathbb{Z}_2^n$ je seznam koeficientů polynomu

$$B(x) = A(x) \cdot G(x)\,.$$

Takto provedené kódování není systematické, tedy prvních k bitů b není identické slovu a. Lze však dosáhnout i systematického kódování při zachování

skutečnosti, že každé kódové slovo $B(x)$ je násobkem $G(x)$. Předpis pro kódování je pak

$$B(x) = A(x) \cdot x^r + (A(x) \cdot x^r) \,\%\, G(x)\,,$$

kde % značí operaci modulo polynom.

Implementace

Jelikož je proces kódování společný všem kódům generovaným polynomem, není realizován samostatnou funkcí v balíčku *BCHCode*. Ke kódování je nutné využít balíčku *PolynomialCode*, konkrétně funkce `PolyEncode`.

Příklad 2 *Nesystematické zakódování informačního slova binárním BCH kódem se zaručenou kódovou vzdáleností $\delta = 5$ sestaveným pomocí tělesa $GF(2^4)$.*

```
In[ ]:= K = BCHCode[4, 5, NoPrint → True];
        PolyEncode[K["G(x)"], {0, 1, 0, 0, 1, 1, 0}, ListLength → K["n"]]

            B(x) = A(x) · G(x)

Out[ ]= {0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0}
```

Nejprve je definován kód K pomocí funkce `BCHCode` *se zakázaným dodatečným výpisem. Následně je funkci* `PolyEncode` *předán generující polynom kódu $G(x)$, který je jedním z klíčů v asociaci reprezentující kód K. Kódováno je informační slovo* 0100110*, výstupem je odpovídající slovo kódové. Parametr* `ListLength` *udává délku kódového slova, ta je rovněž klíčem v asociaci reprezentující kód. Je použito výchozí nesystematické kódování, což dokládá vzorec v dodatečném výpisu.*

4.4.3 Oprava a dekódování

Oprava u binárních BCH kódů sestává ze tří kroků. Nejprve jsou nalezeny syndromy přijatého slova, následně je na základě těchto syndromů zkonstruován tzv. lokátor, neboli mnohočlen pozic, a na závěr jsou na základě lokátoru určeny pozice chyb. Různé algoritmy mohou volit různá konkrétní provedení těchto úkonů, všechny se ale drží naznačeného schématu.

Výpočet syndromů lze realizovat dvěma způsoby. Buď $c \in \mathbb{Z}_2^n$ přijaté slovo, které lze zároveň považovat za koeficienty polynomu $C(x) \in \mathbb{Z}_2[x]$. První způsob je z matice H standardním výpočtem

$$s = c \cdot H^\top\,,$$

kde $s \in \mathbb{Z}_2^{m(\delta-1)}$ lze interpretovat jako $(\delta - 1)$ dílčích syndromů $s_1, \ldots, s_{\delta-1}$ z $GF(2^m)$ tak, že každá m-tice bitů s popisuje jeden prvek $GF(2^m)$. Druhým způsobem je výpočet z polynomu $C(x)$, kde pro každý dílčí syndrom s_i platí

$$s_i = C(\alpha^i)\,,$$

kde α je tentýž primitivní prvek tělesa $GF(2^m)$, který byl použit při konstrukci kódu. V obou případech s_i značí dílčí syndrom vypočtený na základě prvku α^i. Jsou-li všechny dílčí syndromy rovny nulovému prvku $GF(2^m)$, přijaté slovo je bez chyb.

Pro opravu ν chyb je lokátor polynom

$$\Lambda(\xi) = \lambda_\nu \xi^\nu + \ldots + \lambda_2 \xi^2 + \lambda_1 \xi + \lambda_0 \,,$$

kde $\lambda_\nu, \ldots, \lambda_0 \in GF(2^m)$. Pro opravu nejvýše tří chyb je možné použít explicitní vzorce pro výpočet koeficientů $\lambda_\nu, \ldots, \lambda_0$ z dílčích syndromů. Konkrétně následujícím způsobem, kde operace dělení značí standardně násobení inverzním prvkem v tělese $GF(2^m)$ a symbol 1 jeho jednotkový prvek.

Oprava 1 chyby	*Oprava až 2 chyb*	*Oprava až 3 chyb*
$\lambda_0 = 1$	$\lambda_0 = 1$	$\lambda_0 = 1$
$\lambda_1 = s_1$	$\lambda_1 = s_1$	$\lambda_1 = s_1$
	$\lambda_2 = \frac{s_3 + s_1^3}{s_1}$	$\lambda_2 = \frac{s_1^2 \cdot s_3 + s_5}{s_3 + s_1^3}$
		$\lambda_3 = (s_3 + s_1^3) + s_1 \cdot \lambda_2$

Pro opravu většího množství chyb musí být pro výpočet koeficientů lokátoru $\Lambda(\xi)$ použit složitější algoritmus. Například Peterson–Gorensten–Zierler. Ten nejprve nalezne nejvyšší ν takové, že daný kód dokáže opravit ν chyb a zároveň platí, že determinant $det(\Theta_\nu) = 0$, pro matici Θ_ν nad $GF(2^m)$ rozměrů $\nu \times \nu$ sestavenou z dílčích syndromů jako

$$\Theta_\nu = \begin{pmatrix} s_1 & s_2 & \ldots & s_\nu \\ s_2 & s_3 & \ldots & s_{\nu+1} \\ \vdots & \vdots & \ddots & \vdots \\ s_\nu & s_{\nu+1} & \ldots & s_{2\nu-1} \end{pmatrix} .$$

Následně jsou určeny koeficienty lokátoru $\Lambda(\xi)$. Koeficient λ_0 je vždy roven 1. Koeficienty $\lambda_\nu, \ldots, \lambda_1$ jsou dopočteny jako

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

Po sestavení lokátoru $\Lambda(\xi)$ následuje hledání jeho kořenů $\xi_1, \ldots, \xi_\nu$. Pro každý kořen $\xi_i \in GF(2^m)$ je nalezeno takové přirozené číslo $p_i \in \{1, \ldots, n\}$, že $\xi_i = \alpha^{p_i}$. Každé p_i pak udává jednu chybu na p_i-tém bitů slova c, číslováno odpředu od 1.

Postup dekódování opraveného slova $C'(x)$ je stejně jako kódování společný s ostatními kódy generovanými polynomem a existuje ve dvou variantách pro systematický a nesystematický kód. Dekódované slovo $d \in \mathbb{Z}_2^k$ je seznamem koeficientů polynomu

$$D(x) = C'(x) \,/\, G(x) \,,$$

v případě nesystematického kódu. V případě systematického, označíme-li stupeň $G(x)$ jako r, platí

$$D(x) = C'(x) \,/\, x^r \,.$$

Operace / značí dělení polynomů. V obou případech je za předpokladu správné opravy zaručeno, že $C'(x)$ je násobkem $G(x)$. U nesystematického dekódování tudíž dělení proběhne beze zbytku. U systematického dekódování není zaručeno, že $C'(x)$ je násobkem x^r, zbytek po dělení je však ignorován. V podstatě jde o zahození nejnižších koeficientů $C'(x)$, které odpovídají paritním bitům.

Implementace

Opravu provádí funkce `BCHCorrect` mající dva povinné poziční parametry. Jsou jimi kód reprezentovaný asociací a přijaté slovo zadané buď jako $c \in \mathbb{Z}_2^n$ nebo jako polynom $C(x)$ nad $\mathbb{Z}_2$ v proměnné x stupně nejvýše $n-1$. Volitelným třetím pozičním parametrem je některá z funkcí začínající na `GFTo*` z balíčku *CommonGFCode* pro zobrazování prvků konečného tělesa v dodatečném výpisu.

Pro opravu je potřeba stanovit, kolik chyb zadaný kód dokáže opravit. Jedním z parametrů kódu je zaručená kódová vzdálenost δ, z níž lze získat také odhad na minimální počet opravitelných chyb jako $\lfloor(\delta-1)/2\rfloor$. Minimální proto, že skutečná kódová vzdálenost daného kódu může být ve skutečnosti vyšší než δ. Jelikož funkce nemá žádnou další informaci o skutečné kódové vzdálenosti, je za počet opravitelných chyb považováno právě toto minimum. Může tak dojít například k situaci, kdy u kódu s kódovou vzdáleností 7 je možné opravit jen 2 chyby, protože byl konstruován jako kód se zaručenou kódovou vzdáleností $\delta = 5$. To lze napravit zkonstruováním téhož kódu se zaručenou kódovou vzdáleností $\delta = 7$.

Vyjde-li najevo, že je kód použitelný pro opravu nejvýše 3 chyb, jsou koeficienty lokátoru $\Lambda(\xi)$ hledány explicitními vzorci, ve kterých figurují pouze liché dílčí syndromy. V tom případě jsou vypočteny jen tyto syndromy, a to pomocí předpisu

$$s_i = C(\alpha^i) \,.$$

Primitivní prvek α tělesa $GF(2^m)$, který byl použit při konstrukci kódu, je jednou z hodnot v asociaci reprezentující daný kód. Nebylo-li přijaté slovo zadáno jako polynom, je na polynom převedeno. Dosazení do polynomu za x je provedeno systémovou `ReplaceAll`. Vyhodnocení polynomu, jehož koeficienty jsou typu `Integer`, po dosazení prvku konečného tělesa za x proběhne korektně i bez žádných další úprav. Vyjde-li najevo, že kód dokáže opravovat 4 a více chyb, jsou stejným způsobem vypočteny všechny syndromy od s_1 až po $s_{\delta-1}$, včetně těch sudých.

Výpočet koeficientů explicitními vzorci je jednoduchou aplikací základních operací nad konečným tělesem. Při použití Peterson–Gorensten–Zierler algoritmu je nutné hledat determinanty matic nad $GF(2^m)$, počítat jejich inverze

a součiny. S využitím balíčku *FiniteFields* pro reprezentaci prvků $GF(2^m)$ lze všechny tyto operace provést standardními systémovými funkcemi `Det`, `Inverse` a `Dot`.

Hledání kořenů $\Lambda(\xi)$ je prováděno hrubou silou postupným dosazováním za ξ všech prvků $GF(2^m)$ od α^1 až do α^n, podobně jako při výpočtu syndromů. Ze vzorce pro délku kódu $n = 2^m - 1$ plyne, že tímto postupem dojde k vyzkoušení všech nenulových prvků $GF(2^m)$. Přijaté slovo je na závěr opraveno invertováním bitů na pozicích i takových, že α^i je kořenem $\Lambda(\xi)$.

Příklad 3 *Oprava přijatého slova binárního BCH kódu se zaručeno kódovou vzdáleností $\delta = 5$ sestaveného pomocí tělesa $GF(2^4)$. Slovo nese dvě chyby na 3. a posledním bitu.*

```
In[ ]:= K = BCHCode[4, 5, NoPrint → True];
        BCHCorrect[K, {0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1}, GFToBinaryString]

        GF(2^4), δ = 5, n = 15, k = 7

        c = {0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1}
        C(x) = x^13 + x^11 + x^10 + x^9 + x^7 + x^6 + x^2 + x + 1

        Syndrome:
```

i	α^i	$s_i = C(\alpha^i)$
1	0010	1110
3	1000	1101

```
        Locator polynomial Λ(ξ) = λ2 ξ^2 + λ1 ξ + λ0 :
            λ0 = 0001
            λ1 = s1 = 1110
            λ2 = (s3 + s1^3)/ s1 = 1111

            → Λ(ξ) = 1111 ξ^2 + 1110 ξ + 0001
            → roots: ξ1 = 1000, ξ2 = 0001

        Error positions p1, p2:
            ξ1 = 1000 = α^3    →    p1 = 3
            ξ2 = 0001 = α^15    →    p2 = 15

        Correction:
            c = {0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1}
            e = {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1}
            c'= {0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0}
Out[ ]= {0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0}
```

Nejprve je definován kód K pomocí `BCHCode` *se zakázaným dodatečným výpisem. Následně je tento předán funkci* `BCHCorrect` *spolu s přijatým slovem c a funkcí* `GFToBinaryString`*, která způsobí zobrazení prvků konečného tělesa v dodatečném výpisu binárními řetězci.*

Výstupem je opravené slovo c'. Dodatečný výpis nejprve shrnuje základní parametry zadaného kódu a zobrazuje přijaté slovo c v obou formách zápisu.

Následuje tabulka syndromů. Zde jsou uvedeny pouze ty liché, neboť pro kód s $\delta = 5$ jsou koeficienty lokátoru hledány explicitními vzorci. Nalezení lokátoru je rozepsáno hned za tabulkou syndromů. Poté jsou vypsány kořeny lokátoru a jejich převod na pozice chyb. V závěru je na základě pozic jednotlivých chyb sestaveno chybové slovo e a sečteno se slovem c, čímž je jeho oprava hotova.

Pro dekódování není v balíčku *BCHCode* dedikovaná funkce. Používá se `PolyDecode` z balíčku *PolynomialCode* jakožto protějšek `PolyEncode` zmiňované v sekci o kódování.

Příklad 4 *Nesystematické dekódování opraveného slova binárního BCH kódu se zaručenou kódovou vzdáleností $\delta = 5$ sestaveného pomocí tělesa $GF(2^4)$.*

```
In[ ]:= K = BCHCode[4, 5, NoPrint → True];
        PolyDecode[K["G(x)"], {0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0}, ListLength → K["k"]]
            D(x) = C'(x) / G(x)
Out[ ]= {0, 1, 0, 0, 1, 1, 0}
```

Opět je nejprve vytvořen kód K s pomocí `BCHCode` *se zakázaným dodatečným výpisem. Funkci* `PolyDecode` *je předán generující polynom $G(x)$, který je jedním z parametrů v asociaci reprezentující kód, spolu s opraveným slovem c'. Výstupem je dekódované slovo d. Parametr* `ListLength` *udává jeho požadovanou délku. Pro tu je použita dimenze kódu k jakožto jedna z hodnot v asociaci reprezentující kód. Proběhne výchozí nesystematické dekódování, což dokládá vzorec v dodatečném výpisu.*

4.5 Součinové kódy

Součin kódů označuje metodu kombinace dvou lineárních kódů k vytvoření kódu nového. Uvažovány jsou zde pouze binární lineární kódy. Související funkce jsou implementovány v balíčku *ProductCode* a ukázky jejich použití se nachází v notebooku `SoucinoveKody.nb`.

Nejprve je v sekci 4.5.1 popsána konstrukce kódu, následně v 4.5.2 kódování a na závěr v 4.5.3 oprava a dekódování.

Informace čerpány z [8] a [9, s. 79].

4.5.1 Konstrukce kódu

Mějme lineární (n_1, k_1) kód K_1 a další lineární (n_2, k_2) kód K_2. Součinem těchto dvou kódů získáme takový kód K, že jeho délka n a dimenze k splňují

$$n = n_1 \cdot n_2 \qquad \text{a} \qquad k = k_1 \cdot k_2 \,.$$

Slovo $w \in \mathbb{Z}_2^n$ je kódovým slovem, pokud po jeho vepsání po řádcích do matice $W \in \mathbb{Z}_2^{n_2 \times n_1}$ platí, že každý řádek W délky n_1 je kódovým slovem kódu K_1 a každý sloupec matice W délky n_2 je kódovým slovem kódu K_2.

Označíme-li dále δ_1 kódovou vzdálenost kódu K_1 a δ_2 kódovou vzdálenost kódu K_2, pak pro kódovou vzdálenost C_{dist} výsledného kódu K platí

$$C_{dist} = \delta_1 \cdot \delta_2 \,.$$

Implementace

Kódování, oprava a dekódování součinových kódů jsou založeny na opakované aplikaci kódovacích, opravných a dekódovacích procedur dílčích kódů K_1 a K_2. Z hlediska implementace tak bylo důležité zpřístupnit tyto operace spolu se základními informacemi o dílčích kódech funkcím realizujícím kódování nebo opravu a dekódování nad součinovým kódem.

Za tímto účelem byla vytvořena funkce `PartialProductCode`, která uspořádá informace o daném dílčím kódu do podoby, v níž je lze předat dalším funkcím. Má 5 povinných parametrů. Pro kód K_1 je například tím prvním délka kódu n_1, druhým jeho dimenze k_1. Třetím parametrem je kódovací funkce, tedy funkce mající jediný parametr – informační slovo délky k_1, která vrátí odpovídající kódové slovo kódu K_1 délky n_1. Čtvrtým parametrem funkce `PartialProductCode` je obdoba kódovací funkce provádějící opravu slova délky n_1 a jeho následné dekódování na slovo délky k_1. Posledním parametrem je řetězec sloužící jako pojmenování daného dílčího kódu. Pro kód K_2 by byl postup naprosto identický.

Výstupem funkce je asociace shromažďující zadané informace. Vedle jejího vytvoření provádí navíc jednoduchou kontrolu zadaných funkcí pro kódování a opravu s dekódováním. Kontrola je provedena zakódováním nulového informačního slova délky k_1 a porovnáním výsledku s nulovým kódovým slovem délky n_1. V případě, že se neshodují, je uživatel upozorněn chybovou zprávou. Obdobný test s nulovými slovy je proveden i u funkce realizující opravu s dekódováním. Nulové slovo musí být součástí každého lineárního kódu, tudíž i při neznámém kódu by oprava neměla nikdy selhat.

Příklad 1 *Vytvoření dílčího kódu na základě Hammingova* $(7, 4)$*-kódu pro použití v součinovém kódu.*

```
In[•]:= hammingEnc[wA_] := MatrixEncode[HammingCodeG[{7, 4}], wA]
        hammingCorDec[wC_] := MatrixDecode[HammingCodeG[{7, 4}], MatrixCorrect[HammingCodeH[{7, 4}], wC]]
        PartialProductCode[7, 4, hammingEnc, hammingCorDec, "Hamming (7,4)-code"]

Out[•]= <|type → PartialProductCode, n → 7, k → 4, GF → 2,
          name → Hamming (7,4)-code, enc → hammingEnc, cor&dec → hammingCorDec|>
```

Nejprve je definována dvojice funkcí. Funkce `hammingEnc` *zakóduje informační slovo délky 4 generující maticí Hammingova* $(7, 4)$*-kódu. Druhá funkce* `hammingCorDec` *provede opravu přijatého slova délky 7 pomocí syndromu vypočteného z kontrolní matice kódu a poté opravené slovo dekóduje pomocí matice generující.*

Tyto dvě funkce jsou následně spolu s délkou a dimenzí kódu a také jeho textovým popisem předány funkci `PartialProductCode`*, která zadané údaje uspořádá do asociace.*

4.5.2 Kódování

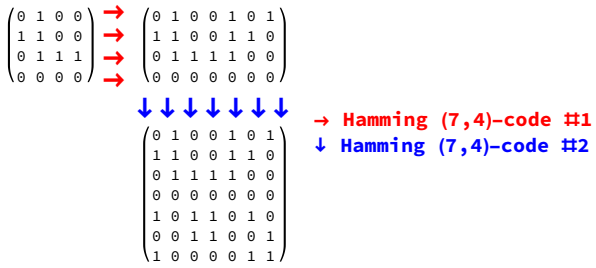
Kódování informačního slova $a \in \mathbb{Z}_2^k$ součinem kódů K_1 a K_2 je prováděno přepsáním a po řádcích do matice rozměrů $k_2 \times k_1$ a následným zakódováním každého řádku pomocí kódu K_1. Tím vznikne matice rozměrů $k_2 \times n_1$. Poté je každý ze sloupců této matice zakódován kódem K_2. Výsledná matice je rozměrů $n_2 \times n_1$. Kódové slovo $b \in \mathbb{Z}_2^n$ je z matice opět získáno po řádcích.

Implementace

Kódování provádí funkce `ProductEncode` mající 3 parametry. Nejdříve to je dvojice dílčích kódů K_1 a K_2 vytvořených pomocí `PartialProductCode` a na závěr informační slovo a. Neodpovídá-li délka informačního slova součinu dimenzí zadaných kódů, je funkce ukončena chybovou zprávou. Vedle kódovacích metod jednotlivých kódů funkce používá pouze základní operace nad maticemi a seznamy.

Příklad 2 *Zakódování informačního slova součinem dvou Hammingových* $(7,4)$*-kódů.*

```
In[ ]:= hammingEnc[wA_] := MatrixEncode[HammingCodeG[{7, 4}], wA]
        hammingCorDec[wC_] := MatrixDecode[HammingCodeG[{7, 4}], MatrixCorrect[HammingCodeH[{7, 4}], wC]]
        K1 = PartialProductCode[7, 4, hammingEnc, hammingCorDec, "Hamming (7,4)-code #1"];
        K2 = PartialProductCode[7, 4, hammingEnc, hammingCorDec, "Hamming (7,4)-code #2"];

        ProductEncode[K1, K2, {0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0}]
```




```
Out[ ]= {0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1}
```

Nejdříve jsou definovány funkce pro kódování a opravu s dekódováním Hammingova $(7,4)$*-kódu. Pomocí nich a funkce* `PartialProductCode` *je následně vytvořena dvojice dílčích kódů* K_1 *a* K_2*.*

Tyto kódy jsou předány funkci `ProductEncode` *spolu s informačním slovem délky 16. Jejím výstupem je odpovídající kódové slovo. Dodatečný výpis zachycuje postup kódování. Od přepsání informačního slova do matice, přes*

aplikaci kódu K_1 na její řádky, až k finální matici odpovídající kódovému slovu. V tomto případě jsou oba dílčí kódy systematické, takže finální matice obsahuje tu původní jako podmatici.

4.5.3 Oprava a dekódování

Opravu a dekódování přijatého slova $c \in \mathbb{Z}_2^n$ lze provést podobně jako kódování přepsáním do matice $n_2 \times n_1$ a následnou opravou a dekódováním každého ze sloupců pomocí kódu K_2, čímž vznikne matice rozměrů $k_2 \times n_1$. Každý její řádek je pak opraven a dekódován kódem K_1. Výsledná matice je rozměrů $k_2 \times k_1$.

Tímto postupem se však nemusí podařit opravit všechny chyby, které by teoreticky na základě kódové vzdálenosti C_{dist} měly být opravitelné. Dokonce může nastat situace, že se například na jednom sloupci slova c zapsaného do matice sejde více chyb, než jaká je korekční schopnost kódu K_2. V takovém případě může oprava daného sloupce, jakožto i celého slova c, selhat.

Implementace

Funkce `ProductCorrectAndDecode` provádí opravu a dekódování součinových kódů. Má 3 povinné parametry – dílčí kódy K_1 a K_2 vytvořené pomocí `PartialProductCode` a přijaté slovo c. Volitelným čtvrtým parametrem je původní odeslané slovo b, které neobsahuje žádnou chybu. S jeho znalostí je možné v dodatečném výpisu zvýraznit propagaci chyb jednotlivými kroky opravy.

Pro opravu a dekódování jednotlivých sloupců a řádků jsou použity funkce předané jako položky v asociacích reprezentujících dílčí kódy. Kromě nich jsou použity pouze základní operace nad maticemi a seznamy. Jak bylo zmíněno výše, může dojít k situaci, kdy oprava dílčím kódem selže. Pokud odpovídající opravná a dekódovací funkce oznámí selhání chybovou zprávou, je tato odchycena systémovou `Check` a výsledný sloupec, či řádek je nahrazen symboly `?` v dodatečném výpisu.

Propagaci chyb je možné sledovat dekódováním chybového slova e, které je dopočteno ze zadaných slov c a b. Funkce `ProductCorrectAndDecode` tak v případě, že jí je předáno i původní odeslané slovo b, provádí dekódování dvou slov najednou – c a e. S tím, že zobrazeno je pouze dekódování slova c, jeho bity jsou ale podbarveny na základě slova e v odpovídající fázi. Ve všech fázích opravy a dekódování platí, že c má chybné bity právě na těch bitech, kde odpovídající bity slova e jsou rovny 1.

Příklad 3 *Oprava a dekódování přijatého slova součinem dvou Hammingových $(7, 4)$-kódů.*

Nejprve jsou definovány funkce pro kódování a opravu s dekódováním Hammingova $(7, 4)$-kódu. Pomocí nich a funkce `PartialProductCode` *je následně*

vytvořena dvojice dílčích kódů K_1 a K_2. Následně je definováno původní odeslané slovo b a přijaté slovo c nesoucí 4 chyby. Obě slova jsou délky 49. Tyto údaje jsou předány funkci `ProductCorrectAndDecode`*, která vrátí opravené a dekódované slovo součinu kódů K_1 a K_2.*

```
In[•]:= hammingEnc[wA_] := MatrixEncode[HammingCodeG[{7, 4}], wA]
        hammingCorDec[wC_] := MatrixDecode[HammingCodeG[{7, 4}], MatrixCorrect[HammingCodeH[{7, 4}], wC]]
        K1 = PartialProductCode[7, 4, hammingEnc, hammingCorDec, "Hamming (7,4)-code #1"];
        K2 = PartialProductCode[7, 4, hammingEnc, hammingCorDec, "Hamming (7,4)-code #2"];

        wordB = {0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1,
           1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1};
        wordC = {0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1,
           1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1};
        ProductCorrectAndDecode[K1, K2, wordC, wordB]
```

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

↓ Hamming (7,4)-code #2
→ Hamming (7,4)-code #1

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

```
Out[•]= {0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0}
```

Dodatečný výpis rozepisuje průběh opravy a dekódování. Díky tomu, že bylo zadáno i slovo b, je možné sledovat vývoj chyby. Podbarvené bity v první matici ukazují 4 chyby na přijatém slově c. Kódová vzdálenost Hammingova $(7,4)$-kódu je 3, kódová vzdálenost součinu dvou těchto kódů je $3 \cdot 3 = 9$, což by mělo umožnit 4 chyby opravit. V tomto příkladě však vidíme, že i finální matice obsahuje jednu chybu, tedy oprava nebyla úspěšná. Hammingův $(7,4)$-kód umožňuje opravit právě jednu chybu. Sejde-li se tudíž na jednom sloupci první matice více chyb, budou chyby přeneseny i do odpovídajícího sloupce druhé matice. Podobné pravidlo platí pro řádky mezi druhou a třetí maticí.

4.6 Konvoluční kódy

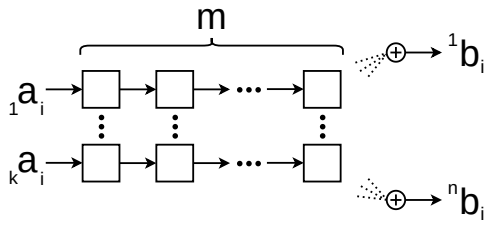
Konvoluční kódy jsou binární kódy dané sekvenčním obvodem. Jsou lineární, ale ne blokové, různá kódová slova mohou být různě dlouhá. Průběh kódování lze vedle obvodu popsat také diskrétní konvolucí. Související funkce jsou implementovány v balíčku *ConvolutionalCode*, ukázky jejich použití se nachází v notebooku `KonvolucniKody.nb`.

Nejprve je v sekci 4.6.1 popsána konstrukce kódu, následně v 4.6.2 kódování a na závěr v 4.6.3 oprava a dekódování.

Informace čerpány z [15] a [7, s. 287–349].

4.6.1 Konstrukce kódu

Pro $n, k, m \in \mathbb{N}$, kde n a k jsou typicky nízká a $k < n$, je konvoluční kód dán sekvenčním obvodem s k vstupy, n výstupy a hloubkou paměti rovnou m. Obvod se skládá z k posuvným m-bitových registrů a n sčítaček modulo 2, jak znázorňuje diagram 4.1. Takový kód označujeme jako konvoluční (n, k, m)-kód. Poměr k/n nazýváme mírou kódu.



Obrázek 4.1: Schéma kodéru konvolučního kódu

Konvoluční kódy se od blokových kódů liší tím, že výstup kodéru není dán pouze jeho vstupem v daném čase, ale i předchozími m vstupy. Vstupní sekvence

$$a = a_1\, a_2\, a_3\, a_4\, \ldots$$

je členěna na slova a_i délky k. Ve chvíli, kdy je na vstupu slovo a_i, je na výstupu slovo b_i délky n. Tato slova pak tvoří výstupní sekvenci

$$b = b_1\, b_2\, b_3\, b_4\, \ldots$$

Každý bit slova b_i je dán součtem vybraných bitů slov a_i až a_{i-m}. Pokud některé z těchto slov neexistuje, jsou jeho bity považovány za nulové.

Označme dále jednotlivé bity slov a_i a b_i jako

$$a_i = {}_1a_i\ {}_2a_i\ \ldots\ {}_ka_i \qquad \text{resp.} \qquad b_i = {}^1b_i\ {}^2b_i\ \ldots\ {}^nb_i\,.$$

To, které bity slov a_i až a_{i-m} jsou sčítány pro získání bitu ${}^vb_i$, udávají takzvané odezvy na impulzy, v anglické literatuře *impulse responses*. Každý konvoluční (n, k, m)-kód je plně charakterizován $k \cdot n$ odezvami na impulzy délky $m + 1$ bitů

$${}^v_ug \;=\; {}^v_ug_0\ {}^v_ug_1\ \ldots\ {}^v_ug_m \qquad \text{pro} \qquad u \in \{1, \ldots, k\},\ v \in \{1, \ldots, n\}\,,$$

kde u-tý bit slova a_{i-w} je součástí sumy při výpočtu v-tého bitu slova b_i právě tehdy, když $w \leq m$ a bit ${}^v_ug_w = 1$. Každý obvod výše popsaného tvaru lze jednoznačně popsat odezvami na impulzy a každá korektní sada odezev na impulzy jednoznačně určuje obvod. Alternativní formou zápisu impulzu ${}^v_ug$ je polynom v proměnné D

$${}^v_uG(D) \;=\; {}^v_ug_0 \cdot 1 + \ {}^v_ug_1 \cdot D \ + \ \ldots \ + \ {}^v_ug_m \cdot D^m\,.$$

Výstup obvodu je v daném okamžiku dán vstupem a obsahem posuvných registrů. Lze jej proto také chápat jako konečný automat typu Mealy. Jeho stav udává km bitů v posuvných registrech, má tedy 2^{km} stavů. Vstupní abecedu tvoří všechny k-tice nad $\mathbb{Z}_2$ a výstupní abecedu n-tice nad $\mathbb{Z}_2$, ne nutně všechny. Z každého stavu vede 2^k přechodů.

Z automatu lze mimo jiné určit tzv. volnou vzdálenost kódu $d_{free} \in \mathbb{N}$, která udává minimální vzdálenost dvou různých kódových slov. Pro určení vzdálenosti dvou slov různé délky se to kratší doplní nulami na délku toho delšího. Hodnotu d_{free} lze určit jako nejmenší Hammingovu váhu nenulového kódového slova. Nebo také jako minimální nenulovou váhu cesty z počátečního stavu automatu zpět do počátečního stavu. Kde za váhu cesty je považována suma Hammingových vah přes všechny výstupní symboly vyprodukované danými přechody. A počáteční stav je stav odpovídající všem bitům v posuvných registrech rovným 0. Volná vzdálenost d_{free} je důležitá pro určování korekční schopnosti kódu. Ta roste s rostoucí volnou vzdáleností. Přesný počet opravitelných chyb však z volné vzdálenosti určit nelze a silně závisí také na metodě opravy.

Implementace

Tvorbu konvolučních kódů provádí funkce `ConvolutionalCode`. Neexistuje však žádná jednoduchá metoda, jak sestavovat kódy s dobrými vlastnostmi. Většinou jsou hledány počítačově. Funkce `ConvolutionalCode` tak umožňuje jak definovat vlastní konvoluční kód, tak využít některého z předdefinovaných kódů čerpaných z [7, s. 330–331].

Vlastní kód lze zadat hloubkou paměti m a maticí M rozměrů $k \times n$, kde každý prvek M_{uv} udává odezvu ${}^{v}_{u}g$ zadanou pomocí binárního řetězce délky $m + 1$.

Totožný formát popisu je využit i pro uložení předdefinovaných kódů v databázi. Pro jejich zadání je však dodržen formát zápisu z [7], kde jsou odezvy zapsány v osmičkové soustavě. Převod je proveden při inicializaci databáze. Ta obsahuje pro každou z mír $n/k \in \{1/4, 1/3, 1/2, 2/3, 3/4\}$ minimálně pět různých kódů s různou hloubkou paměti m. Předáním některého z těchto zlomků funkci `ConvolutionalCode` dojde k vypsání všech dostupných kódů s danou mírou. Jednotlivé řádky tabulky jsou číslovány od jedné. Vybrání konkrétního kódu z tabulky je možné přidáním ID řádku mezi argumenty funkce.

Příklad 1 *Přehled předdefinovaných konvolučních kódů míry* 2/3.

Pro výpis stačí funkci `ConvolutionalCode` *předat odpovídající zlomek. U každého kódu je uvedeno ID řádku sloužící pro výběr daného kódu, hloubka paměti m, odezvy na impulzy zapsané jak binárně, tak polynomy v proměnné D. V posledním sloupci je uvedena volná kódová vzdálenost d_{free} rovněž převzatá ze zdroje [7].*

In[•]:= **ConvolutionalCode[2 / 3]**

Use ConvolutionalCode[rate , ID] where ID is one of the following to get more information about the given code.

ID	m	Impulse Resp. – binary	Impulse Resp. – polynomial	d_{free}
1	1	$\begin{matrix}j\\i\end{matrix}g=\begin{pmatrix}11 & 01 & 11\\01 & 10 & 10\end{pmatrix}_{ij}$	$\begin{matrix}j\\i\end{matrix}G(D)=\begin{pmatrix}1+D & D & 1+D\\D & 1 & 1\end{pmatrix}_{ij}$	3
2	2	$\begin{matrix}j\\i\end{matrix}g=\begin{pmatrix}100 & 010 & 110\\001 & 100 & 111\end{pmatrix}_{ij}$	$\begin{matrix}j\\i\end{matrix}G(D)=\begin{pmatrix}1 & D & 1+D\\D^2 & 1 & 1+D+D^2\end{pmatrix}_{ij}$	4
3	2	$\begin{matrix}j\\i\end{matrix}g=\begin{pmatrix}111 & 001 & 100\\010 & 101 & 111\end{pmatrix}_{ij}$	$\begin{matrix}j\\i\end{matrix}G(D)=\begin{pmatrix}1+D+D^2 & D^2 & 1\\D & 1+D^2 & 1+D+D^2\end{pmatrix}_{ij}$	5
4	3	$\begin{matrix}j\\i\end{matrix}g=\begin{pmatrix}1100 & 0110 & 1110\\0011 & 1000 & 1111\end{pmatrix}_{ij}$	$\begin{matrix}j\\i\end{matrix}G(D)=\begin{pmatrix}1+D & D+D^2 & 1+D+D^2\\D^2+D^3 & 1 & 1+D+D^2+D^3\end{pmatrix}_{ij}$	6
5	3	$\begin{matrix}j\\i\end{matrix}g=\begin{pmatrix}1101 & 0110 & 1101\\0110 & 1101 & 1111\end{pmatrix}_{ij}$	$\begin{matrix}j\\i\end{matrix}G(D)=\begin{pmatrix}1+D+D^3 & D+D^2 & 1+D+D^3\\D+D^2 & 1+D+D^3 & 1+D+D^2+D^3\end{pmatrix}_{ij}$	7

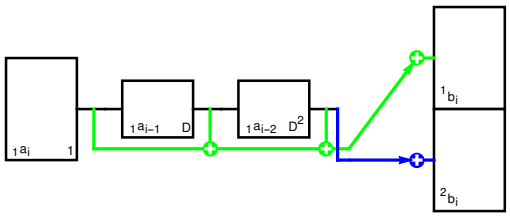
Pro lepší názornost byla implementována speciální funkce, která na základě odezev na impulzy vykreslí odpovídající sekvenční obvod pomocí systémové `Graphics`. Podobně pro zobrazení kódu v podobě konečného automatu byla implementována dedikovaná funkce založená na systémové `Graph` pro vykreslování grafů. Uzly grafu jsou pojmenovány dle hodnot v jednotlivých posuvných registrech v souladu s [15] tak, že například pro 2 registry obsahující bity 100 a 011 je název stavu *S1:6*, kde *S* je univerzální označení pro stav, 1 je decimální zápis binárního 100 po převrácení a 6 je decimální zápis binárního 011 taktéž v opačném pořadí. Vstupní a výstupní symboly jednotlivých přechodů automatu je možné zobrazit umístěním kurzoru myši nad danou hranu v grafu.

Pro uživatelem zadaný konvoluční kód byla vytvořena jednoduchá funkce snažící se v 5 vteřinách prozkoumat všechny cesty z počátečního stavu zpět do počátečního stavu a nalézt nejnižší nenulovou váhu takové cesty pro určení volné vzdálenosti d_{free}. Funkce provádí prohledávání grafu do hloubky bez žádných dalších vylepšení. Pokud se do 5 vteřin nepovede volnou vzdálenost stanovit, zůstává neznámá.

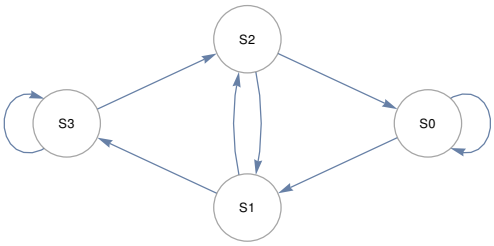
Příklad 2 *Definice vlastního konvolučního kódu.*

Kód je zadán hloubkou paměti 2 a maticí odezev na impulzy. Výstupem je asociace reprezentující daný kód. Obsahuje mimo jiné délku výstupního bloku n, délku vstupního bloku k, hloubku paměti m, odpovědi na impulzy nebo volnou vzdálenost d_{free}. V tomto případě se ji podařilo v intervalu 5 vteřin nalézt, je rovna 4. V dodatečném výpisu jsou nejprve zrekapitulovány základní parametry kódu, poté jsou vypsány odezvy v obou formách zápisu. Následně je zobrazeno schéma kodéru a jeho reprezentace konečným automatem. Ze schématu kodéru je dobře patrná souvislost mezi odezvami na impulzy a výběrem bitů pro jednotlivé součty.

```
In[ ]:= ConvolutionalCode[2, ("111" "001")]

Rate, memory order and free distance:
  k/n = 1/2        m = 2        d_free = 4

Impulse responses - binary:
  1g = 111       2g = 001

Impulse responses - polynomial in D:
  1G(D) = 1 + D + D^2       2G(D) = D^2

Encoder circuit:
```




```
Encoder state machine:
```




```
Out[ ]= <|type → ConvolutionalCode, n → 2, k → 1, GF → 2, m → 2, IR → {{111, 001}}, d_free → 4|>
```

4.6.2 Kódování

Konvolučním (n, k, m)-kódem lze zakódovat libovolnou posloupnost bitů rozdělitelnou na slova délky k. Mějme vstupní sekvenci $a = a_1 \dots a_p$ délky p slov, tedy pk bitů. Jejím zakódováním získáme výstupní sekvenci $b = b_1 \dots b_q$ délky q slov, neboli qn bitů, kde

$$q = p + m\,.$$

Tedy za výstupní slovo se považuje veškerý výstup obvodu od vstupu prvního slova a_1 až po to poslední a_p a následné vyprázdnění všech m fází posuvných registrů s uvažovaným nulovým vstupem.

Kódování lze samozřejmě provádět odpovídajícím sekvenčním obvodem, případně jeho simulací nebo odpovídajícím konečným automatem. Existují však ještě další 2 přístupy, jak na kódování pohlížet.

Tím prvním je přístup založený na násobení polynomů, což také odpovídá diskrétní konvoluci seznamů jejich koeficientů. Mějme pro každé $u \in \{1, \dots, k\}$ podsekvenci ${}_u a$ délky p bitů vzniklou ze vstupní sekvence a vybrání u-tého

bitu z každého slova a_1 až a_p a odpovídající polynom ${}_uA(D)$ získaný jako

$$\begin{aligned} {}_ua &= {}_ua_1\ {}_ua_2\ {}_ua_3\ \ldots\ {}_ua_p\,, \\ {}_uA(D) &= {}_ua_1 \cdot 1 + {}_ua_2 \cdot D + \ldots + {}_ua_p \cdot D^{p+1}\,. \end{aligned}$$

Dále nechť pro každé $v \in \{1, \ldots, n\}$ je ${}^vb$ podsekvence délky q vytvořené z výstupní sekvence b vybráním v-tého bitu z každého slova b_1 až b_q. A dále opět odpovídají polynom ${}^vB(D)$ daný jako

$$\begin{aligned} {}^vb &= {}^vb_1\ {}^vb_2\ {}^vb_3\ \ldots\ {}^vb_q\,, \\ {}^vB(D) &= {}^vb_1 \cdot 1 + {}^vb_2 \cdot D + \ldots + {}^vb_q \cdot D^{q+1}\,. \end{aligned}$$

Pak každou výstupní podsekvenci ${}^vb$ lze získat ze vstupních podsekvencí ${}_ua$ a odezev na impulzy ${}^v_ug$ aplikací diskrétní konvoluce $*$. Podobně všechny odpovídající polynomy ${}^vB(D)$ lze získat ze součinů polynomů ${}_uA(D)$ a ${}^v_uG(D)$. Konkrétně sumou přes všechna $u \in \{1, \ldots, k\}$ jako

$${}^vb = \sum_{u=1}^{k} {}_ua * {}^v_ug \qquad \text{resp.} \qquad {}^vB(D) = \sum_{u=1}^{k} {}_uA(D) \cdot {}^v_uG(D)\,.$$

Finální výstupní sekvence b je z podsekvencí ${}^vb$ získána jejich proložením. Nad polynomy lze proložení zapsat pomocí sumy

$$B(D) = \sum_{v=1}^{n} {}^vB(D^n) \cdot D^{u-1}\,,$$

kde dosazení D^n za D provádí „roztažení“ podsekvence ${}^vB(D)$ a násobení D^{u-1} její „posun“.

Posledním možným přístupem ke kódování je maticové násobením. Mějme pro každé $i \in \{1, \ldots, m+1\}$ binární matici G_i rozměrů $k \times n$ získanou vybráním i-tého bitu z každé odezvy na impulzy ${}^v_ug$ tak, že

$$(G_i)_{uv} = {}^v_ug_i\,.$$

Pak zakódování sekvence a délky p slov lze provést násobením maticí $b = a' \cdot G$, kde a' značí sekvenci a doplněnou nulovými slovy na délku $q = p + m$ slov a G je binární matice rozměrů $kq \times nq$ sestavená z podmatic G_i jako

$$G = \begin{pmatrix} G_1 & G_2 & \ldots & G_{m+1} & O & \ldots & O \\ O & G_1 & \ldots & G_m & G_{m+1} & \ldots & O \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ O & O & \ldots & O & O & \ldots & G_1 \end{pmatrix},$$

kde O značí nulovou matici rozměrů $k \times n$. V zápisu podmaticemi je každý řádek G získán posunem toho předchozího o jednu pozici doprava, každý sloupec i řádek je tvořen q podmaticemi.

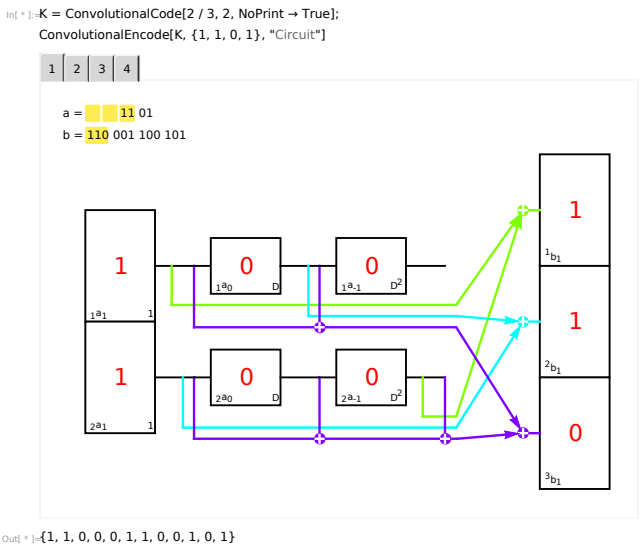
Implementace

Kódování u konvolučních kódů provádí funkce `ConvolutionalEncode` s dvěma povinnými pozičními parametry – kódem reprezentovaným asociací a informační sekvencí a zadanou seznamem bitů. Volitelně lze jako třetí parametr uvést některý z řetězců `"Circuit"`, `"Graph"`, `"Polynomial"` nebo `"Matrix"` pro výběr způsobu kódování. Pokud délka zadaného a v bitech není dělitelná k, je kódování ukončeno s chybou.

Pro volbu `"Circuit"` je kódování prováděno simulací daného sekvenčního obvodu. Stav obvodu je reprezentován k-ticí celých čísel, jejichž byty tvoří obsah k posuvných registrů. Aktualizace stavu a výpočet výstupu jsou prováděny základními bitovými operacemi, například systémovými `BitShiftRight` nebo `BitAnd`, spolu se součtem modulo 2 systémovou `Mod`.

Dodatečný výstup zobrazuje schéma obvodu spolu s aktuálními hodnotami daných bitů. Pro každý krok je systémovou `Graphics` vytvořen samostatný obrázek. Ty jsou zorganizovány do záložek pomocí systémové `TabView`. Proházet obrázky lze umístěním kurzoru myši nad požadovanou záložku.

Příklad 3 *Zakódování sekvence 1101 konvolučním $(3, 2, 2)$-kódem simulací sekvenčního obvodu.*

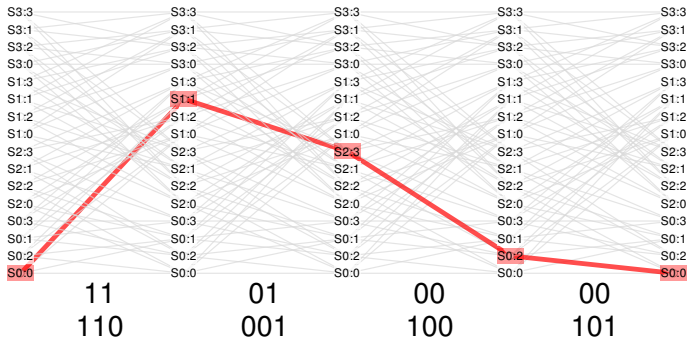


Nejprve je vybrán druhý kód s mírou $2/3$ z databáze. Ten je předán funkci `ConvolutionalEncode` *spolu se vstupní sekvencí odpovídající dvěma slovům délky 2 a řetězcem udávajícím metodu kódování. Výstupem funkce je výstupní sekvence daná seznamem bitů, který odpovídá čtyřem slovům délky 3. V levém horním rohu je zobrazena vstupní a výstupní sekvence rozdělená na slova. Ta slova, která jsou aktuálně zpracovávána jsou zvýrazněna podbarvením.*

Pro ukázání kódování konečným automatem je použit tzv. *trellis* diagram. Jedná se o multipartitní graf, jehož každá partita obsahuje všechny stavy konečného automatu, pokaždé však v jiný časový okamžik. Hrany z partity reprezentující čas t vedou pouze do partity reprezentující čas $t + 1$. Pomocí *trellis* diagramu tak je možné zachytit průběh kódování přehledně v jediném grafu.

Příklad 4 *Zakódování sekvence 1101 konvolučním* $(3, 2, 2)$*-kódem pomocí konečného automatu.*

```
In[•]:= K = ConvolutionalCode[2/3, 2, NoPrint → True];
        ConvolutionalEncode[K, {1, 1, 0, 1}, "Graph"]
```




```
Out[•]= {1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1}
```

Nejprve je vybrán druhý kód s mírou $2/3$ *z databáze. Ten je předán funkci* `ConvolutionalEncode` *spolu se vstupní sekvencí a řetězcem udávajícím metodu kódování. Výstupem funkce je výstupní sekvence daná seznamem bitů.*

Dodatečný výstup zobrazuje trellis diagram. Nejlevější partita odpovídá času 0, automat se nachází v počátečním stavu $S0 : 0$*. Tento stav je zvýrazněn červenou barvou. Prvním vstupním symbolem automatu je slovo 11, což způsobí přechod do stavu* $S1 : 1$ *a vypsání slova 110 na výstup. Podobně pro další kroky. Vstupní sekvence je doplněna* $m = 2$ *nulovými slovy, čímž je zajištěn návrat do počátečního stavu. Vstupní a výstupní symboly jednotlivých přechodů jsou zobrazeny pod diagramem.*

Kódování pomocí polynomů využívá pouze základní operace nad polynomy nad $\mathbb{Z}_2$. Aby byla patrná souvislost mezi polynomiálním zápisem s operací součin polynomů a zápisem pomocí binárních sekvencí s operací diskrétní konvoluce, jsou v dodatečném výpisu uvedeny oba postupy vedle sebe. Pro výpočet diskrétní konvoluce je však rovněž použit součin polynomů.

Příklad 5 *Zakódování sekvence 1101 konvolučním* $(3, 2, 2)$*-kódem s využitím polynomů v proměnné* D*.*

Opět je nejprve vybrán druhý kód s mírou $2/3$ *z databáze. Ten je předán funkci* `ConvolutionalEncode` *spolu se vstupní sekvencí a řetězcem udávajícím metodu kódování. Výstupem funkce je výstupní sekvence daná seznamem bitů.*

```
In[ ]:= K = ConvolutionalCode[2/3, 2, NoPrint → True];
        ConvolutionalEncode[K, {1, 1, 0, 1}, "Polynomial"]
```

Polynomial	Binary
${}^1_1G(D) = 1$ ${}^2_1G(D) = D$ ${}^3_1G(D) = 1 + D$ ${}^1_2G(D) = D^2$ ${}^2_2G(D) = 1$ ${}^3_2G(D) = 1 + D + D^2$	${}^1_1g = 100$ ${}^2_1g = 010$ ${}^3_1g = 110$ ${}^1_2g = 001$ ${}^2_2g = 100$ ${}^3_2g = 111$
$A(D) = 1 + D + D^3$ ${}_1A(D) = 1$ ${}_2A(D) = 1 + D$ ${}^1B(D) = {}_1A(D)\cdot{}^1_1G(D) + {}_2A(D)\cdot{}^1_2G(D)$ $= 1 + D^2 + D^3$ ${}^2B(D) = {}_1A(D)\cdot{}^2_1G(D) + {}_2A(D)\cdot{}^2_2G(D)$ $= 1$ ${}^3B(D) = {}_1A(D)\cdot{}^3_1G(D) + {}_2A(D)\cdot{}^3_2G(D)$ $= D + D^3$ $B(D) = {}^1B(D^3)\cdot D^0 + {}^2B(D^3)\cdot D^1 + {}^3B(D^3)\cdot D^2$ $= 1 + D + D^5 + D^6 + D^9 + D^{11}$	$a = 1\ 1\ 0\ 1$ ${}_1a = 1\ 0$ ${}_2a = 1\ 1$ ${}^1b = {}_1a * {}^1_1g + {}_2a * {}^1_2g$ $= 1\ 0\ 1\ 1$ ${}^2b = {}_1a * {}^2_1g + {}_2a * {}^2_2g$ $= 1\ 0\ 0\ 0$ ${}^3b = {}_1a * {}^3_1g + {}_2a * {}^3_2g$ $= 0\ 1\ 0\ 1$ $b = 1\ 1\ 0\ 0\ 0\ 1\ 1\ 0\ 0\ 1\ 0\ 1$

```
Out[ ]= {1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1}
```

V levé části dodatečného výpisu je postup zapsán polynomy, v pravé části binárními sekvencemi. Tam jsou nejprve vypsány odezvy na impulzy ${}^v_u g$. Následně je vstupní sekvence a rozdělena na podsekvence ${}_u a$, což je doplněno barevným zvýrazněním. Následně jsou pomocí konvoluce a sčítání vypočteny jednotlivé výstupní podsekvence ${}^v b$ a ty na závěr zkombinovány do finální výstupní sekvence b. Barevná zvýraznění pro podsekvence ${}_u a$ a ${}^v b$ jsou navzájem nezávislá. Levá strana uvádí ekvivalentní kroky v polynomiálním zápisu.

Podbarvování bitů v dodatečném výpisu systémovou `Highlighted` je ve značné míře využito také při kódování pomocí matice. Konkrétně k tomu, aby bylo zdůrazněno, které bity matice G pochází z které odezvy na impulzy. Samotné zakódování je pak provedeno funkcí `MatrixEncode` z balíčku *CommonCode*.

Příklad 6 *Zakódování sekvence 1101 konvolučním $(3, 2, 2)$-kódem pomocí matice sestavené z odezev na impulzy.*

Znovu je nejprve vybrán druhý kód s mírou $2/3$ z databáze. Ten je předán funkci `ConvolutionalEncode` *spolu se vstupní sekvencí a řetězcem udávajícím metodu kódování. Výstupem funkce je výstupní sekvence daná seznamem bitů.*

V dodatečném výpisu jsou nejdříve zobrazeny odezvy na impulzy uspořádané do matice v tom pořadí, ve kterém jsou z nich vybírány matice G_i. Matice G_i sice nejsou explicitně vypsány, díky barevnému podbarvení jsou však dobře patrné.

```
In[ ]:= K = ConvolutionalCode[2/3, 2, NoPrint → True];
        ConvolutionalEncode[K, {1, 1, 0, 1}, "Matrix"]

        ¹₁g = 1 0 0      ²₁g = 0 1 0      ³₁g = 1 1 0
        ¹₂g = 0 0 1      ²₂g = 1 0 0      ³₂g = 1 1 1

        G =
        ⎛ 1 0 1 0 1 1 0 0 0 0 0 0 ⎞
        ⎜ 0 1 1 0 0 1 1 0 1 0 0 0 ⎟
        ⎜ 0 0 0 1 0 1 0 1 1 0 0 0 ⎟
        ⎜ 0 0 0 0 1 1 0 0 1 1 0 1 ⎟
        ⎜ 0 0 0 0 0 0 1 0 1 0 1 1 ⎟
        ⎜ 0 0 0 0 0 0 0 1 1 0 0 1 ⎟
        ⎜ 0 0 0 0 0 0 0 0 0 1 0 1 ⎟
        ⎝ 0 0 0 0 0 0 0 0 0 0 1 1 ⎠

        a = 11 01    →    Expanded to: a' = 11 01 00 00
        b = a'·G = 110 001 100 101
Out[ ]= {1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1}
```

4.6.3 Oprava a dekódování

Existuje velké množství algoritmů pro opravu konvolučních kódů a jejich variant. Zde jsou popsány dvě metody – Viterbiho algoritmus a sekvenční dekódování – tak, jak je uvádí [15].

Mějme konvoluční (n, k, m)-kód a přijatou sekvenci c tvořenou q slovy délky n. Obě metody využívají tzv. *trellis* diagram založený na konečném automatu daného kódu. Konečný automat má 2^{km} stavů. *Trellis* diagram je pro účely opravy a dekódování slova délky q tvořen $q + 1$ vrstvami, z nichž každá obsahuje 2^{km} vrcholů odpovídajících všem stavům konečného automatu. Jednotlivé vrstvy jsou uspořádány zleva doprava a číslovány od 0 do q. Všechny hrany z libovolné vrstvy i směřují pouze do následující vrstvy $i + 1$, pokud tato existuje. Pro každý z přechodů konečného automatu je mezi sousedními vrstvami hrana spojující vrcholy odpovídající daným stavům automatu.

Cílem opravných algoritmů je nalézt takovou cestu z počátečního stavu v nulté vrstvě do počátečního stavu v poslední q-té vrstvě, že odpovídající přechody generují opravenou sekvenci $c' = c'_1 \dots c'_q$ pro $c'_i \in \mathbb{Z}_2^n$. Odpovídající vstupní sekvence $d = d_1 \dots d_q$ pro $d_i \in \mathbb{Z}_2^k$ je považována za výslednou dekódovanou sekvenci.

Viterbiho algoritmus sestavuje cestu v *trellis* diagramu tak, že pro všechna $i \in \{1, \dots, q\}$ ohodnotí každou hranu mezi vrstvou $i - 1$ a i počtem bitů, ve kterých se shodují výstupní symbol daného přechodu a přijaté slovo c_i. Nazveme-li ohodnocením cesty součet ohodnocení všech jejich hran, vybere Viterbiho algoritmus takovou cestu z počátečního stavu nulté vrstvy do počá-

tečního stavu poslední vrstvy, že její ohodnocení je maximální. Odpovídající sekvence c' tak má s přijatou sekvencí c maximum schodných bitů.

Pro sekvenční dekódování je nutné znát maximální opravitelný počet chyb t. Algoritmus pak vrátí první cestu z počátečního stavu v nulté vrstvě do počátečního stavu v q-té vrstvě takovou, že odpovídající sekvence c' se od c liší v nejvýše v t bitech. Algoritmus startuje v počátečním stavu nulté vrstvy a cesty sestavuje v závislosti na tom, která z těchto situací nastane:

1. Existuje-li mezi přechody z aktuálního stavu v i-té vrstvě takový, že jeho výstupní symbol je roven c_{i+1}, je tento zařazen do cesty a zpracování pokračuje cílovým stavem ve vrstvě $i + 1$.

2. Neplatí-li 1. a přitom existují mezi přechody z aktuálního stavu takové, že jejich zařazením do cesty nedojde k překročení limitu t chyb, jsou prozkoumány všechny tyto možnosti v pořadí od minimálního počtu přidaných chyb.

3. Neplatí-li 1. ani 2. jsou z cesty odebírány hrany, dokud cesta nekončí ve stavu s ještě neprozkoumanými přijatelnými přechody.

Implementace

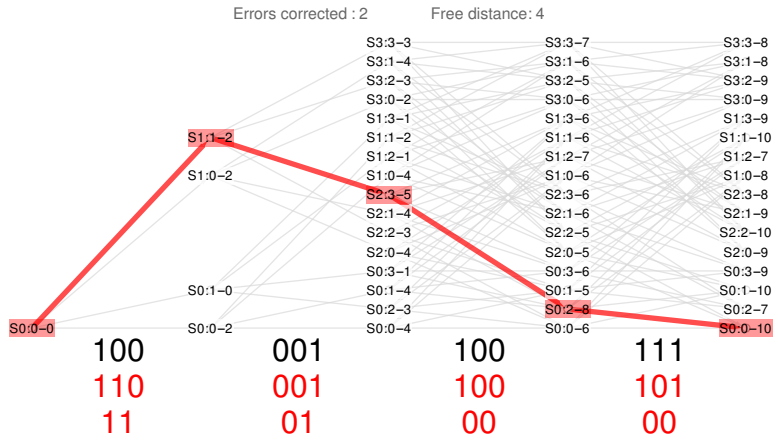
Opravu a dekódování realizuje funkce `ConvolutionalCorrectAndDecode` mající 3 povinné parametry – kód reprezentovaný asociací, přijatou sekvenci zadanou seznamem bitů a řetězec udávající, který algoritmus má být použit. Povolené řetězce jsou `"Viterbi"` a `"Sequential"`.

Hledání cesty s nejvyšším ohodnocením u Viterbiho algoritmu je prováděno postupným ohodnocováním vrcholů v jednotlivých vrstvách. Každému vrcholu je přiřazeno maximální ohodnocení cesty vedoucí do něj z počátečního stavu nulté vrstvy. Ohodnocování vrcholů dosud neohodnocené vrstvy využívá výsledků z předchozí vrstvy, dokud není stanoveno i ohodnocení všech vrcholů v poslední vrstvě. Pokaždé, když je nalezena nová nejlepší cesta do některého z vrcholů, je zároveň uložen i vrchol předchozí vrstvy, skrze nějž bylo tohoto maxima dosaženo. Využitím těchto údajů je na závěr možné zrekonstruovat výslednou cestu spojující počáteční stavy v nulté a poslední vrstvě.

Příklad 7 *Viterbiho algoritmus pro opravu a dekódování přijaté sekvence u konvolučního* $(3, 2, 2)$*-kódu.*

Nejprve je vybrán 2 kód s mírou $2/3$ *z databáze předdefinovaných kódů. Následně je tento použit pro opravu přijaté sekvence s chybou na 2. a předposledním bitu. Výstupem je dekódovaná sekvence 1101.*

```
In[ ]:= K = ConvolutionalCode[2/3, 2, NoPrint → True];
        ConvolutionalCorrectAndDecode[K, {1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1}, "Viterbi"]
```




```
Out[ ]= {1, 1, 0, 1}
```

Dodatečný výpis zobrazuje trellis diagram se zvýrazněnou nejlepší cestou. Jednotlivé vrcholy jsou pojmenovány dle stavů konečného automatu, součástí názvu je však také ohodnocení daného vrcholu – poslední hodnota za pomlčkou. Nad diagramem je uvedeno, kolik chyb bylo opraveno. Pod diagramem je černou barvou vypsána přijatá sekvence c, červenou opravená sekvence c' a dekódovaná sekvence d v tomto pořadí. Z dekódované sekvence bylo odstraněno závěrečných $m = 2$ nulových slov, ty totiž sloužily pouze jako výplň při kódování, aby došlo k vyprázdnění posuvných registrů.

Pro účely sekvenčního dekódování je nutné stanovit maximální přijatelný počet bitů, v nichž se může opravená sekvence lišit od té zadané. Funkci `ConvolutionalCorrectAndDecode` je proto nutné tuto hodnotu zadat prostřednictvím pojmenovaného parametru `ErrorLimit`. Hledání uspokojivé cesty je realizováno rekurzivním průchodem *trellis* diagramu z počátečního stavu nulté vrstvy dle uvedených pravidel. Není-li pro stanovený maximální počet chyb nalezena žádná cesta, je oprava ukončena s chybovou hláškou. Je však zajištěno, aby i v tomto případě byl zobrazen *trellis* diagram znázorňující všechny prozkoumané cesty.

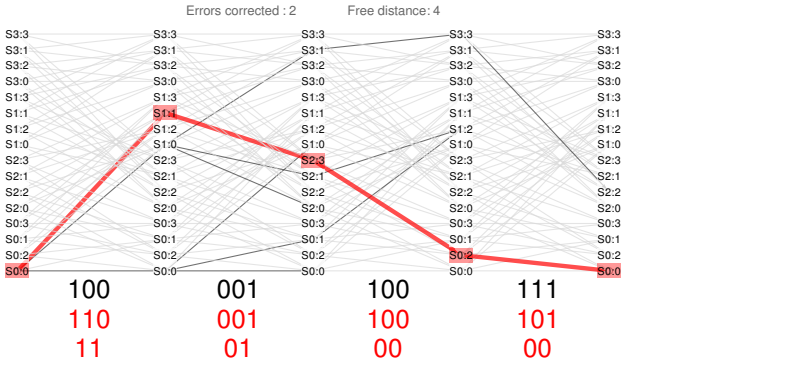
Příklad 8 *Sekvenční dekódování přijaté sekvence u konvolučního* $(3, 2, 2)$*-kódu.*

Znovu je nejprve vybrán kód K s mírou $2/3$ z databáze. Ten je následně předán funkci `ConvolutionalCorrectAndDecode` *spolu s přijatou sekvencí s chybami na 2. a předposledním bitu, řetězcem udávajícím výběr algoritmu a pojmenovaným parametrem* `ErrorLimit`*, který říká, že má být akceptována první cesta s méně než dvěma bity odlišnými od zadané sekvence. Výstupem je dekódovaná sekvence 1101.*

Součástí dodatečného výpisu je opět trellis diagram. Prozkoumané cesty jsou v něm zvýrazněny tmavější šedou a finální cesta červenou barvou. Počet

opravených chyb je vypsána nad diagramem. Pod diagramem je vypsána přijatá sekvence c, opravená sekvence c' a dekódovaná sekvence d v tomto pořadí. Dekódovaná sekvence byla zkrácena o $m = 2$ výplňových nulových slov.

```
K = ConvolutionalCode[2/3, 2, NoPrint → True];
ConvolutionalCorrectAndDecode[K, {1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1}, "Sequential", ErrorLimit → 2]
```



Out[•]= {1, 1, 0, 1}

4.7 Nebinární BCH kódy

Bose–Chaudhuri–Hocquenghem (zkráceně BCH) kódy jsou kódy generované polynomem pro opravu volitelného množství chyb. Tato podkapitola je věnována nebinárním BCH kódům, binární BCH kódy jsou popsány v podkapitole 4.4. Speciální třídě nebinárních BCH kódů, RS kódům, je taktéž věnována samostatná podkapitola 4.8. Související funkce jsou implementovány v balíčku *BCHCode* a ukázky jejich použití se nachází v notebooku `BCHNebinarniKody.nb`.

Nejprve je v sekci 4.7.1 popsána konstrukce kódu, následně v 4.7.2 kódování a na závěr v 4.7.3 oprava a dekódování.

Informace čerpány z [13] a [7, s. 170–177].

4.7.1 Konstrukce kódu

Princip tvorby nebinárního BCH kódu je velmi podobný tvorbě binárního BCH kódu. Místo tělesa $GF(2^m)$ však využívá rozšířené těleso $GF((2^{m_1})^{m_2})$, tedy těleso tvořené polynomy nad $GF(2^{m_1})$ modulo ireducibilní definiční polynom stupně m_2. Samotná kódová slova jsou nad $GF(2^{m_1})$, stejně tak generující polynom nebo kontrolní matice kódu. Délka kódu n se neudává v bitech, nýbrž v prvcích $GF(2^{m_1})$. Každý prvek $GF(2^{m_1})$ lze zapsat posloupností m_1 bitů, a každé kódové slovo je tedy posloupností $m_1 \cdot n$ bitů. Dílčí posloupnosti m_1 bitů jsou pak označovány jako slabiky.

Dojde-li při přenosu k invertování libovolného počtu bitů v jedné slabice, hovoříme stále o chybě velikosti 1. Korekční schopnost kódu je udávána v počtu slabik, který umožňuje opravit. Proto o nebinárních kódech mluvíme také jako

o kódech pro opravu slabik. A i zaručená kódová vzdálenost δ je udávána ve slabikách. Za vzdálenost dvou slov stejné délky považujeme počet slabik, ve kterých se liší.

Nebinární BCH kód je dán rozšířeným konečným tělesem $GF((2^{m_1})^{m_2})$ pro $m_1, m_2 \in \mathbb{N}, m_1 > 1$ a požadovanou zaručenou kódovou vzdáleností $\delta \in \mathbb{N}$. Výsledný kód je délky $n = 2^{m_1 m_2} - 1$ se skutečnou kódovou vzdáleností $C_{dist} \geq \delta$. Speciálním případem BCH kódů jsou kódy RS, pro něž $m_2 = 1$. Těm je věnována samostatná podkapitola 4.8. V této podkapitole jsou dále uvažovány jen kódy s $m_2 > 1$.

Mějme stejně jako u binárních kódů nějaký primitivní prvek α tělesa $GF((2^{m_1})^{m_2})$ a označme $M_{\#i}(x)$ pro $i \in \mathbb{N}_0$ minimální polynom prvku α^i. Pro prvek $GF((2^{m_1})^{m_2})$ jsou koeficienty jeho minimálního polynomu z $GF(2^{m_1})$. Generující polynom nad $GF(2^{m_1})$ pak získáme jako

$$G(x) = LCM(M_{\#1}(x), M_{\#2}(x), \ldots, M_{\#\delta-2}(x), M_{\#\delta-1}(x)),$$

kde LCM značí nejmenší společný násobek daných minimálních polynomů. Jeho stupeň r udává redundanci kódu, a tedy i jeho dimenzi $k = n - r$. Oba tyto údaje jsou dány ve slabikách.

Kontrolní matice je v souladu s binárními BCH kódy nejprve sestavena z prvků $GF((2^{m_1})^{m_2})$ jako

$$H = \begin{pmatrix} (\alpha^1)^{n-1} & (\alpha^1)^{n-2} & \ldots & (\alpha^1)^2 & \alpha^1 & 1 \\ (\alpha^2)^{n-1} & (\alpha^2)^{n-2} & \ldots & (\alpha^2)^2 & \alpha^2 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ (\alpha^{\delta-1})^{n-1} & (\alpha^{\delta-1})^{n-2} & \ldots & (\alpha^{\delta-1})^2 & \alpha^{\delta-1} & 1 \end{pmatrix}.$$

A následně je každý prvek rozepsán na sloupec m_2 prvků z $GF(2^{m_1})$. Výsledkem je matice nad $GF(2^{m_1})$ rozměrů $m(\delta - 1) \times n$.

Matice H může obsahovat lineárně závislé řádky. Matice vzniklá jejich odstraněním bude stále splňovat definici kontrolní matice, tedy že pro libovolné kódové slovo c je syndrom $s = c \cdot H^\top$ nulový. Mají-li však být z H počítány všechny dílčí syndromy $s_1, \ldots, s_{\delta-1}$ nutné k opravě slova c, musí být lineárně závislé řádky zachovány.

Implementace

Konstrukci nebinárních BCH kódů provádí stejná funkce jako konstrukci binárních BCH kódů – `BCHCode`. Má dva povinné poziční parametry, specifikaci tělesa a δ. Podle specifikace tělesa rozezná, zda má jít o binární, či nebinární kód. Rozšířené těleso $GF((2^{m_1})^{m_2})$ je možné zadat seznamem dvou celých čísel m_1 a m_2 nebo ireducibilním polynomem nad $GF(2^{m_1})$ stupně m_2, který bude pro definici rozšířeného tělesa použit. Je-li zadána pouze dvojice m_1 a m_2, je rozšířené těleso zkonstruováno pomocí výchozího ireducibilního polynomu

nad vnitřním tělesem $GF(2^{m_1})$, pro jehož definici je zvolen rovněž výchozí ireducibilní polynom, tentokrát nad $\mathbb{Z}_2$ stupně m_1.

Pro reprezentaci prvků rozšířených těles je použita jejich vlastní implementace blíže popsaná na straně 15. Umožňuje s prvky rozšířených těles pracovat podobně jako s prvky těles realizovaných systémovým balíčkem *FiniteFields*. Je tak například možné je sčítat, násobit a mocnit běžnými systémovými funkcemi `Plus`, `Times` a `Power` a jim odpovídajícími operátory. Prvek $GF((2^{m_1})^{m_2})$ je vždy v dodatečném výpisu zobrazen jako m_2-tice koeficientů z $GF(2^{m_1})$. Způsob zobrazování prvků $GF(2^{m_1})$ lze ovlivnit třetím pozičním parametrem funkce `BCHCode`, předáním některé z funkcí z *CommonGFCode* začínajících na `GFTo*`.

Je-li pro definici $GF((2^{m_1})^{m_2})$ použit výchozí ireducibilní polynom z databáze, je generujícím prvkem tohoto tělesa vždy prvek $\alpha = 1y$ za předpokladu, že ireducibilní polynom je v proměnné y. 1 značí jednotkový prvek $GF(2^{m_1})$. V opačném případě je primitivní prvek α hledán hrubou silou.

Hledání hrubou silou je prováděno výběrem nějakého $u \in GF((2^{m_1})^{m_2})$ a ověřením, zdali mocniny u generují všech $N = 2^{m_1 m_2} - 1$ nenulových prvků tělesa. Neboli ověřením, jestli je řád prvku u roven N, neboli jestli N je nejmenší přirozené číslo, pro které $u^N = 1$. Řád prvku u musí vždy dělit řád grupy N. Tedy u je primitivním prvkem, pokud pro všechny $i \in \mathbb{N}$ dělitele N kromě N samotného je $u^i \neq 1$. Naopak je-li $u^i = 1$ pro nějaké $i < N$, pak u primitivním prvkem není. Hledání začíná od polynomů stupně 1, pokud mezi nimi není nalezen žádný primitivním prvek, pokračuje se všemi polynomy tvořenými jediným monomem s koeficientem 1. Není-li ani mezi nimi primitivní prvek nalezen, jsou další u voleny náhodně. Multiplikativní grupy rozšířených i nerozšířených těles jsou vždy cyklické, proto je existence nějakého primitivního prvku zaručena. [16]

Minimální polynomy jsou konstruovány pomocí `GFMinimalPolynomial` z balíčku *CommonGFCode* stejně jako u binárních kódů. Způsob hledání jejich nejmenšího společného násobku se také nemění, je použita systémová `DeleteDuplicates` k odstranění duplicitních výskytů všech minimálních polynomů a následně proveden jejich součin.

Příklad 1 *Nebinární BCH kód s zaručenou kódovou vzdáleností $\delta = 5$ sestavený pomocí rozšířeného tělesa $GF((2^2)^2)$.*

Vstupem funkce `BCHCode` *je specifikace tělesa $GF((2^{m_1})^{m_2})$ dvojicí exponentů m_1 a m_2, zaručená kódová vzdálenost δ a funkce* `GFToDecimal` *mající za následek zobrazení prvků $GF(2^2)$ v dodatečném výpisu jako decimálních čísel.*

Výstupem je asociace reprezentující kód. Obsahuje například délku kódu n a jeho dimenzi k, generující polynom $G(x)$, primitivní prvek α, ireducibilní polynom použitý pro definici $GF((2^2)^2)$ nebo kontrolní matici kódu H. V dodatečném výpisu jsou nejprve zrekapitulovány zadané hodnoty. Následně je zobrazena tabulka minimálních polynomů prvků α^1 až $\alpha^{\delta-1}$. Z těch je poté

vypočten generující polynom $G(x)$ a z jeho stupně dimenze kódu k. V závěru je rozepsána konstrukce matice H spolu s přepisem prvků $GF((2^2)^2)$ na sloupce 2 prvků nad $GF(2^2)$.

```
In[•]:= BCHCode[{2, 2}, 5, GFToDecimal]
```

$GF(4^2)$, δ = 5, n = 15

Minimal polynomials:

i	α^i	$M_{i+1}(x)$
1	$\{1, 0\}_{\omega\{a^{+2}\}}$	$1x^2 + 1x + 2$
2	$\{1, 2\}_{\omega\{a^{+2}\}}$	$1x^2 + 1x + 3$
3	$\{3, 2\}_{\omega\{a^{+2}\}}$	$1x^2 + 3x + 1$
4	$\{1, 1\}_{\omega\{a^{+2}\}}$	$1x^2 + 1x + 2$

$G(x) = LCM[M_{i1}(x), M_{i2}(x), M_{i3}(x), M_{i4}(x)]$

$G(x) = M_{i1}(x) \cdot M_{i2}(x) \cdot M_{i3}(x)$

$G(x) = (1x^2 + 1x + 2) \cdot (1x^2 + 1x + 3) \cdot (1x^2 + 3x + 1)$

$G(x) = 1x^6 + 3x^5 + 1x^4 + 1x^3 + 2x^2 + 2x + 1$

$k = n - \deg[G(x)] = 15 - 6 = 9$

$$H = \begin{pmatrix} \alpha^{14} & \alpha^{13} & \alpha^{12} & \alpha^{11} & \alpha^{10} & \alpha^{9} & \alpha^{8} & \alpha^{7} & \alpha^{6} & \alpha^{5} & \alpha^{4} & \alpha^{3} & \alpha^{2} & \alpha & 1 \\ (\alpha^2)^{14} & (\alpha^2)^{13} & (\alpha^2)^{12} & (\alpha^2)^{11} & (\alpha^2)^{10} & (\alpha^2)^{9} & (\alpha^2)^{8} & (\alpha^2)^{7} & (\alpha^2)^{6} & (\alpha^2)^{5} & (\alpha^2)^{4} & (\alpha^2)^{3} & (\alpha^2)^{2} & \alpha^2 & 1 \\ (\alpha^3)^{14} & (\alpha^3)^{13} & (\alpha^3)^{12} & (\alpha^3)^{11} & (\alpha^3)^{10} & (\alpha^3)^{9} & (\alpha^3)^{8} & (\alpha^3)^{7} & (\alpha^3)^{6} & (\alpha^3)^{5} & (\alpha^3)^{4} & (\alpha^3)^{3} & (\alpha^3)^{2} & \alpha^3 & 1 \\ (\alpha^4)^{14} & (\alpha^4)^{13} & (\alpha^4)^{12} & (\alpha^4)^{11} & (\alpha^4)^{10} & (\alpha^4)^{9} & (\alpha^4)^{8} & (\alpha^4)^{7} & (\alpha^4)^{6} & (\alpha^4)^{5} & (\alpha^4)^{4} & (\alpha^4)^{3} & (\alpha^4)^{2} & \alpha^4 & 1 \end{pmatrix}$$

$$H = \begin{pmatrix} \{3,3\}_{\omega\{a^{+2}\}} & \{2,1\}_{\omega\{a^{+2}\}} & \{3,1\}_{\omega\{a^{+2}\}} & \{3,0\}_{\omega\{a^{+2}\}} & \{0,3\}_{\omega\{a^{+2}\}} & \{2,2\}_{\omega\{a^{+2}\}} & \{1,3\}_{\omega\{a^{+2}\}} & \{2,3\}_{\omega\{a^{+2}\}} & \{2,0\}_{\omega\{a^{+2}\}} & \{0,2\}_{\omega\{a^{+2}\}} & \{1,1\}_{\omega\{a^{+2}\}} & \{3,2\}_{\omega\{a^{+2}\}} & \{1,2\}_{\omega\{a^{+2}\}} & \{1,0\}_{\omega\{a^{+2}\}} & \{0,1\}_{\omega\{a^{+2}\}} \\ \{2,1\}_{\omega\{a^{+2}\}} & \{3,0\}_{\omega\{a^{+2}\}} & \{2,2\}_{\omega\{a^{+2}\}} & \{2,3\}_{\omega\{a^{+2}\}} & \{0,2\}_{\omega\{a^{+2}\}} & \{3,2\}_{\omega\{a^{+2}\}} & \{1,0\}_{\omega\{a^{+2}\}} & \{3,3\}_{\omega\{a^{+2}\}} & \{3,1\}_{\omega\{a^{+2}\}} & \{0,3\}_{\omega\{a^{+2}\}} & \{1,3\}_{\omega\{a^{+2}\}} & \{2,0\}_{\omega\{a^{+2}\}} & \{1,1\}_{\omega\{a^{+2}\}} & \{1,2\}_{\omega\{a^{+2}\}} & \{0,1\}_{\omega\{a^{+2}\}} \\ \{3,1\}_{\omega\{a^{+2}\}} & \{2,2\}_{\omega\{a^{+2}\}} & \{2,0\}_{\omega\{a^{+2}\}} & \{3,2\}_{\omega\{a^{+2}\}} & \{0,1\}_{\omega\{a^{+2}\}} & \{3,1\}_{\omega\{a^{+2}\}} & \{2,1\}_{\omega\{a^{+2}\}} & \{2,0\}_{\omega\{a^{+2}\}} & \{3,2\}_{\omega\{a^{+2}\}} & \{0,1\}_{\omega\{a^{+2}\}} & \{3,1\}_{\omega\{a^{+2}\}} & \{2,2\}_{\omega\{a^{+2}\}} & \{2,0\}_{\omega\{a^{+2}\}} & \{3,2\}_{\omega\{a^{+2}\}} & \{0,1\}_{\omega\{a^{+2}\}} \\ \{3,0\}_{\omega\{a^{+2}\}} & \{2,3\}_{\omega\{a^{+2}\}} & \{3,2\}_{\omega\{a^{+2}\}} & \{3,3\}_{\omega\{a^{+2}\}} & \{0,3\}_{\omega\{a^{+2}\}} & \{2,0\}_{\omega\{a^{+2}\}} & \{1,2\}_{\omega\{a^{+2}\}} & \{2,1\}_{\omega\{a^{+2}\}} & \{2,2\}_{\omega\{a^{+2}\}} & \{0,2\}_{\omega\{a^{+2}\}} & \{1,0\}_{\omega\{a^{+2}\}} & \{3,1\}_{\omega\{a^{+2}\}} & \{1,3\}_{\omega\{a^{+2}\}} & \{1,1\}_{\omega\{a^{+2}\}} & \{0,1\}_{\omega\{a^{+2}\}} \end{pmatrix}$$

$$H = \begin{pmatrix} \begin{pmatrix} 3 & 2 & 3 & 3 & 0 & 2 & 1 & 2 & 2 & 0 & 1 & 3 & 1 & 1 & 0 \\ 3 & 1 & 1 & 0 & 3 & 2 & 3 & 3 & 0 & 2 & 1 & 2 & 2 & 0 & 1 \end{pmatrix} \\ \begin{pmatrix} 2 & 3 & 2 & 2 & 0 & 3 & 1 & 3 & 3 & 0 & 1 & 2 & 1 & 1 & 0 \\ 1 & 0 & 2 & 3 & 2 & 2 & 0 & 3 & 1 & 3 & 3 & 0 & 1 & 2 & 1 \end{pmatrix} \\ \begin{pmatrix} 3 & 2 & 2 & 3 & 0 & 3 & 2 & 2 & 3 & 0 & 3 & 2 & 2 & 3 & 0 \\ 1 & 2 & 0 & 2 & 1 & 1 & 2 & 0 & 2 & 1 & 1 & 2 & 0 & 2 & 1 \end{pmatrix} \\ \begin{pmatrix} 3 & 2 & 3 & 3 & 0 & 2 & 1 & 2 & 2 & 0 & 1 & 3 & 1 & 1 & 0 \\ 0 & 3 & 2 & 3 & 3 & 0 & 2 & 1 & 2 & 2 & 0 & 1 & 3 & 1 & 1 \end{pmatrix} \end{pmatrix}$$

```
Out[•]:= ⟨| type → BCHCode, n → 15, k → 9, GF → GF[2, {1, 1, 1}], δ → 5,
  G(x) → x {1, 0}_ω{a^+2} + x^2 {1, 0}_ω{a^+2} + {0, 1}_ω{a^+2} + x^3 {0, 1}_ω{a^+2} + x^4 {0, 1}_ω{a^+2} + x^5 {1, 1}_ω{a^+2}, α → {{0, 1}_ω{a^+2}, 0}_ω{a^+2}, dp → {{0, 1}_ω{a^+2}, {0, 1}_ω{a^+2}, {1, 0}_ω{a^+2}},
  H → {{{1, 1}_ω{a^+2}, {1, 0}_ω{a^+2}, {1, 1}_ω{a^+2}, {1, 1}_ω{a^+2}, 0, {1, 0}_ω{a^+2}, {0, 1}_ω{a^+2}, {1, 0}_ω{a^+2}, {1, 0}_ω{a^+2}, 0, {0, 1}_ω{a^+2}, {1, 1}_ω{a^+2}, {0, 1}_ω{a^+2}, {0, 1}_ω{a^+2}, 0},
    {{1, 1}_ω{a^+2}, {0, 1}_ω{a^+2}, {0, 1}_ω{a^+2}, 0, {1, 1}_ω{a^+2}, {1, 0}_ω{a^+2}, {1, 1}_ω{a^+2}, {1, 1}_ω{a^+2}, 0, {1, 0}_ω{a^+2}, {0, 1}_ω{a^+2}, {1, 0}_ω{a^+2}, {1, 0}_ω{a^+2}, 0, {0, 1}_ω{a^+2}},
    {{1, 0}_ω{a^+2}, {1, 1}_ω{a^+2}, {1, 0}_ω{a^+2}, {1, 0}_ω{a^+2}, 0, {1, 1}_ω{a^+2}, {0, 1}_ω{a^+2}, {1, 1}_ω{a^+2}, {1, 1}_ω{a^+2}, 0, {0, 1}_ω{a^+2}, {1, 0}_ω{a^+2}, {0, 1}_ω{a^+2}, {0, 1}_ω{a^+2}, 0},
    {{0, 1}_ω{a^+2}, 0, {1, 0}_ω{a^+2}, {1, 1}_ω{a^+2}, {1, 0}_ω{a^+2}, {1, 0}_ω{a^+2}, 0, {1, 1}_ω{a^+2}, {0, 1}_ω{a^+2}, {1, 1}_ω{a^+2}, {1, 1}_ω{a^+2}, 0, {0, 1}_ω{a^+2}, {1, 0}_ω{a^+2}, {0, 1}_ω{a^+2}},
    {{1, 1}_ω{a^+2}, {1, 0}_ω{a^+2}, {1, 0}_ω{a^+2}, {1, 1}_ω{a^+2}, 0, {1, 1}_ω{a^+2}, {1, 0}_ω{a^+2}, {1, 0}_ω{a^+2}, {1, 1}_ω{a^+2}, 0, {1, 1}_ω{a^+2}, {1, 0}_ω{a^+2}, {1, 0}_ω{a^+2}, {1, 1}_ω{a^+2}, 0},
    {{0, 1}_ω{a^+2}, {1, 0}_ω{a^+2}, 0, {1, 0}_ω{a^+2}, {0, 1}_ω{a^+2}, {0, 1}_ω{a^+2}, {1, 0}_ω{a^+2}, 0, {1, 0}_ω{a^+2}, {0, 1}_ω{a^+2}, {0, 1}_ω{a^+2}, {1, 0}_ω{a^+2}, 0, {1, 0}_ω{a^+2}, {0, 1}_ω{a^+2}},
    {{1, 1}_ω{a^+2}, {1, 0}_ω{a^+2}, {1, 1}_ω{a^+2}, {1, 1}_ω{a^+2}, 0, {1, 0}_ω{a^+2}, {0, 1}_ω{a^+2}, {1, 0}_ω{a^+2}, {1, 0}_ω{a^+2}, 0, {0, 1}_ω{a^+2}, {1, 1}_ω{a^+2}, {0, 1}_ω{a^+2}, {0, 1}_ω{a^+2}, 0},
    {0, {1, 1}_ω{a^+2}, {1, 0}_ω{a^+2}, {1, 1}_ω{a^+2}, {1, 1}_ω{a^+2}, 0, {1, 0}_ω{a^+2}, {0, 1}_ω{a^+2}, {1, 0}_ω{a^+2}, {1, 0}_ω{a^+2}, 0, {0, 1}_ω{a^+2}, {1, 1}_ω{a^+2}, {0, 1}_ω{a^+2}, {0, 1}_ω{a^+2}}}, EC → at least 2 errors |⟩
```

4.7.2 Kódování

Kódování u nebinárních BCH kódů je prováděno jako u ostatních kódů generovaných polynomem, ať už nad $\mathbb{Z}_2$ nebo nad $GF(2^m)$. A to tak, že je informační slovo a nad $GF(2^{m_1})$ délky k považováno za seznam koeficientů polynomu $A(x)$, který je vynásoben generujícím polynomem $G(x)$ taktéž nad $GF(2^{m_1})$ stupně r. Kódové slovo b nad $GF(2^{m_1})$ délky n je seznam koeficientů polynomu

$$B(x) = A(x) \cdot G(x)\,.$$

Takto provedené kódování není systematické, tedy prvních k slabik b není identické slovu a. Lze však dosáhnout i systematického kódování při zachování skutečnosti, že každé kódové slovo $B(x)$ je násobkem $G(x)$. Předpis pro kódování je pak

$$B(x) = A(x) \cdot x^r + (A(x) \cdot x^r)\,\%\,G(x)\,,$$

kde % značí operaci modulo polynom.

Implementace

Jelikož je proces kódování společný všem kódům generovaným polynomem, není realizován samostatnou funkcí v balíčku *BCHCode*. Ke kódování je nutné využít balíčku *PolynomialCode*, konkrétně funkce `PolyEncode`, která umí zpracovat jak binární, tak nebinární kódy.

Příklad 2 *Nesystematické zakódování informačního slova nebinárním BCH kódem se zaručenou kódovou vzdáleností $\delta = 5$ sestaveným pomocí rozšířeného tělesa $GF((2^2)^2)$.*

```
In[ ]:= K = BCHCode[{2, 2}, 5, NoPrint → True];
        wordA = GFFromDecimal[2, {0, 1, 2, 3, 0, 3, 2, 1, 0}];
        PolyEncode[K["G(x)"], wordA, ListLength → K["n"]] // GFToDecimal

            B(x) = A(x) · G(x)

Out[ ]= {0, 1, 1, 3, 1, 0, 2, 0, 1, 1, 3, 2, 0, 1, 0}
```

Nejprve je definován kód K pomocí funkce `BCHCode` *se zakázaným dodatečným výpisem, následně informační slovo a nad tělesem $GF(2^2)$ daným výchozím ireducibilním polynomem odpovídajících rozměrů. Prvky jsou zadány převodem z decimálního zápisu. Funkci* `PolyEncode` *je předán generující polynom kódu $G(x)$, který je jedním z klíčů v asociaci reprezentující kód K a slovo a. Výstupem je odpovídající slovo kódové. To je zobrazeno taktéž v decimální notaci. Parametr* `ListLength` *udává délku kódového slova, ta je klíčem v asociaci reprezentující kód. Je použito výchozí nesystematické kódování, což dokládá vzorec v dodatečném výpisu.*

4.7.3 Oprava a dekódování

Oprava nebinárních BCH kódů probíhá zpočátku stejně jako i binárních kódů – jsou nalezeny syndromy přijatého slova, následně je na základě těchto syndromů zkonstruován lokátor a na základě lokátoru určeny pozice chyb. Oproti binárním BCH kódům je u těch nebinárních nutné přidat ještě jeden krok určující tvar chyb, neboť chybou je rozuměna chybná slabika, nikoliv chybný bit. A jedna slabika může nabývat více než 2 různých hodnot, takže znalost pozice chyby ještě nedává dostatek informací k její opravě.

Výpočet syndromů lze realizovat oběma způsoby známými z binárních BCH kódů. Buď c přijaté slovo nad $GF(2^{m_1})$ délky n, které lze zároveň považovat za koeficienty polynomu $C(x) \in GF(2^{m_1})[x]$. První způsob je z matice H standardním výpočtem

$$s = c \cdot H^\top ,$$

kde s nad $GF(2^{m_1})$ je délky $m_2(\delta-1)$ a lze jej interpretovat jako $(\delta-1)$ dílčích syndromů $s_1, \ldots, s_{\delta-1} \in GF((2^{m_1})^{m_2})$ tak, že každá m_2-tice slabik s popisuje

jeden prvek $GF((2^{m_1})^{m_2})$. Druhým způsobem je výpočet z polynomu $C(x)$, kde pro každý dílčí syndrom s_i platí

$$s_i = C(\alpha^i)\,,$$

kde α je tentýž primitivní prvek tělesa $GF((2^{m_1})^{m_2})$, který byl použit při konstrukci kódu. Do polynomu nad $GF(2^{m_1})$ je dosazován prvek $\alpha \in GF((2^{m_1})^{m_2})$. To je v pořádku, neboť každý prvek $u \in GF(2^{m_1})$ lze chápat jako polynom

$$0 \cdot y^{m_2-1} + \ldots + 0 \cdot y^2 + 0 \cdot y + u\,,$$

a tedy i jako prvek $GF((2^{m_1})^{m_2})$. V obou případech s_i značí dílčí syndrom vypočtený na základě prvku α^i. Jsou-li všechny dílčí syndromy rovny nulovému prvku $GF((2^{m_1})^{m_2})$, přijaté slovo je bez chyb.

Pro opravu ν chyb je lokátor polynom

$$\Lambda(\xi) = \lambda_\nu \xi^\nu + \ldots + \lambda_2 \xi^2 + \lambda_1 \xi + \lambda_0\,,$$

kde $\lambda_\nu, \ldots, \lambda_0 \in GF((2^{m_1})^{m_2})$. Explicitní vzorce pro výpočet jednotlivých koeficientů používané u binárních BCH kódů pro opravu nejvýše 3 chyb zde není možné použít. Koeficienty jsou i pro nízký počet chyb určovány Peterson–Gorensten–Zierler algoritmem.

Ten nejprve nalezne nejvyšší ν takové, že daný kód dokáže opravit ν chyb a zároveň platí, že determinant $det(\Theta_\nu) = 0$, pro matici Θ_ν nad $GF((2^{m_1})^{m_2})$ rozměrů $\nu \times \nu$ sestavenou z dílčích syndromů jako

$$\Theta_\nu = \begin{pmatrix} s_1 & s_2 & \ldots & s_\nu \\ s_2 & s_3 & \ldots & s_{\nu+1} \\ \vdots & \vdots & \ddots & \vdots \\ s_\nu & s_{\nu+1} & \ldots & s_{2\nu-1} \end{pmatrix}.$$

Následně jsou určeny koeficienty lokátoru $\Lambda(\xi)$. Koeficient λ_0 je vždy roven 1. Koeficienty $\lambda_\nu, \ldots, \lambda_1$ jsou dopočteny jako

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

Po sestavení lokátoru $\Lambda(\xi)$ následuje hledání jeho kořenů $\xi_1, \ldots, \xi_\nu$. Pro každý kořen $\xi_i \in GF((2^{m_1})^{m_2})$ je nalezeno přirozené číslo $p_i \in \{1, \ldots, n\}$ takové, že $\xi_i = \alpha^{p_i}$. Každé p_i pak udává jednu chybu na p_i-té slabice slova c, číslováno odpředu od 1.

Na závěr je pro každou chybu na p_i-té slabice určen její tvar $\epsilon_i \in GF(2^{m_1})$. Ten je dopočten z kořenů $\xi_1, \ldots, \xi_\nu$ a dílčích syndromů $s_1, \ldots, s_\nu$ prostřednictvím matice Ω jako

$$\Omega = \begin{pmatrix} \xi_1^{-1} & \xi_2^{-1} & \ldots & \xi_\nu^{-1} \\ \xi_1^{-2} & \xi_2^{-2} & \ldots & \xi_\nu^{-2} \\ \vdots & \vdots & \ddots & \vdots \\ \xi_1^{-\nu} & \xi_2^{-\nu} & \ldots & \xi_\nu^{-\nu} \end{pmatrix} \quad \text{a} \quad \begin{pmatrix} \epsilon_1 \\ \vdots \\ \epsilon_\nu \end{pmatrix} = \Omega^{-1} \begin{pmatrix} s_1 \\ \vdots \\ s_\nu \end{pmatrix}.$$

Jelikož kořeny ξ_i i dílčí syndromy s_i jsou nad $GF((2^{m_1})^{m_2})$, je i výsledek výše uvedeného součinu matic nad tímto rozšířeným tělesem. Je však zaručeno, že jde pouze o konstantní polynomy, které jsou zároveň prvky $GF(2^{m_1})$.

Oprava je dokončena přičtením všech ϵ_i k odpovídající slabice slova c na pozici p_i při číslování od 1 odpředu.

Postup dekódování opraveného slova $C'(x)$ je stejně jako kódování společný s ostatními kódy generovanými polynomem a existuje ve dvou variantách pro systematický a nesystematický kód. Rozdílem oproti binárním kódům je to, že slova a polynomy nejsou nad $\mathbb{Z}_2$, ale nad slabikami $GF(2^{m_1})$. Dekódované slovo d nad $GF(2^{m_1})$ délky k je seznamem koeficientů polynomu

$$D(x) = C'(x) \,/\, G(x)\,,$$

v případě nesystematického kódu. V případě systematického, označíme-li stupeň $G(x)$ jako r, platí

$$D(x) = C'(x) \,/\, x^r\,.$$

Operace / značí dělení polynomů. V obou případech je za předpokladu správné opravy zaručeno, že $C'(x)$ je násobkem $G(x)$. U nesystematického dekódování tudíž dělení proběhne beze zbytku. U systematického dekódování není zaručeno, že $C'(x)$ je násobkem x^r, zbytek po dělení je však ignorován. V podstatě jde o zahození nejnižších koeficientů $C'(x)$, které odpovídají paritním bitům.

Implementace

Opravu nebinárních BCH kódů provádí funkce `BCHCorrect` stejně jako u binárních kódů. Ze zadaných parametrů rozezná, o kterou variantu kódu jde. Funkce má dva povinné poziční parametry. Jsou jimi kód reprezentovaný asociací a přijaté slovo zadané buď jako slovo c nad $GF(2^{m_1})$ délky n nebo jako polynom $C(x)$ nad $GF(2^{m_1})$ v proměnné x stupně nejvýše $n-1$. Volitelným třetím pozičním parametrem je některá z funkcí začínající na `GFTo*` z balíčku *CommonGFCode* pro zobrazování prvků konečného tělesa v dodatečném výpisu.

Pro opravu je potřeba stanovit, kolik chyb zadaný kód dokáže opravit. To je provedeno stejně jako u binárních kódů ze zaručené kódové vzdálenosti δ jako $\lfloor(\delta-1)/2\rfloor$. I zde může nastat situace, že kód ve skutečnosti umožňuje opravit i více chyb.

Syndromy jsou vypočteny pomocí předpisu

$$s_i = C(\alpha^i)\,.$$

Primitivní prvek α tělesa $GF((2^{m_1})^{m_2})$, který byl použit při konstrukci kódu, je jednou z hodnot v asociaci reprezentující daný kód. Nebylo-li přijaté slovo zadáno jako polynom, je na polynom převedeno. Dosazení do polynomu za x je provedeno systémovou `ReplaceAll`. Aby vyhodnocení polynomu proběhlo

správně, jsou nejprve všechny koeficienty $C(x)$ z $GF(2^{m_1})$ převedeny na konstantní polynomy v $GF((2^{m_1})^{m_2})$.

V Peterson–Gorensten–Zierler algoritmu je nutné hledat determinanty matic nad $GF((2^{m_1})^{m_2})$, počítat jejich inverze a součiny. S využitím vlastní implementace rozšířených konečných těles, která je blíže popsána na straně 3.1.3, lze všechny tyto operace provést standardními systémovými funkcemi `Det`, `Inverse` a `Dot`.

Hledání kořenů $\Lambda(\xi)$ je prováděno hrubou silou postupným dosazováním za ξ všech prvků $GF((2^{m_1})^{m_2})$ od α^1 až do α^n. Ze vzorce pro délku kódu $n = 2^{m_1 m_2} - 1$ plyne, že tímto postupem dojde k vyzkoušení všech nenulových prvků $GF((2^{m_1})^{m_2})$.

Výpočet tvarů chyb ϵ_i nepoužívá oproti Peterson–Gorensten–Zierler algoritmu žádné jiné operace. Pouze je v závěru nutné převést prvky $GF((2^{m_1})^{m_2})$ odpovídající konstantním polynomům na prvky $GF(2^{m_1})$. To je při vnitřní reprezentaci seznamem koeficientů provedeno jednoduše vybráním poslední položky v tomto seznamu.

Příklad 3 *Oprava přijatého slova nebinárního BCH kódu se zaručeno kódovou vzdáleností $\delta = 5$ sestaveného pomocí tělesa $GF((2^2)^2)$. Slovo nese dvě chyby na 2. a předposlední slabice.*

Nejprve je definován kód K pomocí `BCHCode` *se zakázaným dodatečným výpisem, následně přijaté slovo c nad tělesem $GF(2^2)$. Prvky jsou zadány převodem z decimálního zápisu. Funkci* `BCHCorrect` *je předán kód K, přijaté slovo c a funkce* `GFToDecimal`*, která způsobí zobrazení prvků konečného tělesa v dodatečném výpisu decimálními čísly.*

Výstupem je opravené slovo c' nad $GF(2^2)$ zobrazené decimálně. Dodatečný výpis nejprve shrnuje základním parametry zadaného kódu a zobrazuje přijaté slovo c v obou formách zápisu. Následuje tabulka syndromů. Poté je rozepsán výpočet matice Θ_ν, jakožto součást Peterson–Gorensten–Zierler algoritmu. Hned pro $\nu = 2$ je její determinant nenulový, což znamená, že c obsahuje 2 chyby. Z matice Θ_2 jsou dále vypočteny koeficienty lokátoru $\Lambda(\xi)$ a nalezeny jeho kořeny. Z kořenů jsou určeny pozice chyb p_i a v kombinaci se syndromy i jejich tvary ϵ_i. Oba výpočty jsou rozepsány po jednotlivých krocích. V závěru je na základě pozic a tvarů jednotlivých chyb sestaveno chybové slovo e a sečteno se slovem c, čímž je jeho oprava hotova. Vidíme, že byly skutečně identifikovány a opraveny chyby na 2. a 14. (tedy předposlední) slabice.

In[•]:=
```
K = BCHCode[{2, 2}, 5, NoPrint → True];
wordC = GFFromDecimal[2, {0, 3, 1, 3, 1, 0, 2, 0, 1, 1, 3, 2, 0, 0, 0}];
BCHCorrect[K, wordC, GFToDecimal] // GFToDecimal
```

$GF(4^2)$, $\delta = 5$, $n = 15$, $k = 9$

$c = \{0, 3, 1, 3, 1, 0, 2, 0, 1, 1, 3, 2, 0, 0, 0\}$

$C(x) = 3x^{13} + 1x^{12} + 3x^{11} + 1x^{10} + 2x^{8} + 1x^{6} + 1x^{5} + 3x^{4} + 2x^{3}$

Syndrome:

i	α^i	$s_i = C(\alpha^i)$
1	$\{1, 0\}_{GF(4^2)}$	$\{2, 2\}_{GF(4^2)}$
2	$\{1, 2\}_{GF(4^2)}$	$\{0, 2\}_{GF(4^2)}$
3	$\{3, 2\}_{GF(4^2)}$	$\{0, 1\}_{GF(4^2)}$
4	$\{1, 1\}_{GF(4^2)}$	$\{2, 0\}_{GF(4^2)}$

Matrix Θ_v:

Starting from the highest correctable error size $v = 2$:

$$\Theta_2 = \begin{pmatrix} s_1 & s_2 \\ s_2 & s_3 \end{pmatrix} = \begin{pmatrix} \{2, 2\}_{GF(4^2)} & \{0, 2\}_{GF(4^2)} \\ \{0, 2\}_{GF(4^2)} & \{0, 1\}_{GF(4^2)} \end{pmatrix} \quad \rightarrow \quad \det(\Theta_2) = \{2, 1\}_{GF(4^2)}$$

Determinant is non-zero → error size $v = 2$

Locator polynomial $\Lambda(\xi) = \lambda_2 \xi^2 + \lambda_1 \xi + \lambda_0$:

$$\begin{pmatrix} \lambda_2 \\ \lambda_1 \end{pmatrix} = \Theta_2^{-1} \cdot \begin{pmatrix} s_3 \\ s_4 \end{pmatrix} = \begin{pmatrix} \{1, 2\}_{GF(4^2)} & \{2, 3\}_{GF(4^2)} \\ \{2, 3\}_{GF(4^2)} & \{3, 0\}_{GF(4^2)} \end{pmatrix} \cdot \begin{pmatrix} \{0, 1\}_{GF(4^2)} \\ \{2, 0\}_{GF(4^2)} \end{pmatrix} = \begin{pmatrix} \{3, 3\}_{GF(4^2)} \\ \{3, 1\}_{GF(4^2)} \end{pmatrix}$$

→ $\Lambda(\xi) = \{3, 3\}_{GF(4^2)} \xi^2 + \{3, 1\}_{GF(4^2)} \xi + \{0, 1\}_{GF(4^2)}$

→ roots: $\xi_1 = \{1, 2\}_{GF(4^2)}$, $\xi_2 = \{3, 3\}_{GF(4^2)}$

Error positions p_1, p_2:

$\xi_1 = \{1, 2\}_{GF(4^2)} = \alpha^2 \quad \rightarrow \quad p_1 = 2$

$\xi_2 = \{3, 3\}_{GF(4^2)} = \alpha^{14} \quad \rightarrow \quad p_2 = 14$

Error values ϵ_1, ϵ_2:

$$\Omega = \begin{pmatrix} \xi_1^{-1} & \xi_2^{-1} \\ \xi_1^{-2} & \xi_2^{-2} \end{pmatrix} = \begin{pmatrix} \{2, 1\}_{GF(4^2)} & \{1, 0\}_{GF(4^2)} \\ \{3, 0\}_{GF(4^2)} & \{1, 2\}_{GF(4^2)} \end{pmatrix}$$

$$\begin{pmatrix} \epsilon_1 \\ \epsilon_2 \end{pmatrix} = \Omega^{-1} \cdot \begin{pmatrix} s_1 \\ s_2 \end{pmatrix} = \begin{pmatrix} \{2, 0\}_{GF(4^2)} & \{0, 2\}_{GF(4^2)} \\ \{0, 1\}_{GF(4^2)} & \{1, 2\}_{GF(4^2)} \end{pmatrix} \cdot \begin{pmatrix} \{2, 2\}_{GF(4^2)} \\ \{0, 2\}_{GF(4^2)} \end{pmatrix} = \begin{pmatrix} \{0, 2\}_{GF(4^2)} \\ \{0, 1\}_{GF(4^2)} \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$$

Correction:

$c = \{0, 3, 1, 3, 1, 0, 2, 0, 1, 1, 3, 2, 0, 0, 0\}$

$e = \{0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0\}$

$c' = \{0, 1, 1, 3, 1, 0, 2, 0, 1, 1, 3, 2, 0, 1, 0\}$

Out[•]= $\{0, 1, 1, 3, 1, 0, 2, 0, 1, 1, 3, 2, 0, 1, 0\}$

Pro dekódování není v balíčku *BCHCode* dedikovaná funkce. Používá se `PolyDecode` z balíčku *PolynomialCode* jakožto protějšek `PolyEncode` zmiňované v sekci o kódování. Stejně jako `PolyEncode` umí i `PolyDecode` pracovat s binárními i nebinárními kódy.

Příklad 4 *Nesystematické dekódování opraveného slova nebinárního BCH kódu se zaručenou kódovou vzdáleností $\delta = 5$ sestaveného pomocí rozšířeného tělesa $GF((2^2)^2)$.*

```
In[ ]:= K = BCHCode[{2, 2}, 5, NoPrint → True];
        wordCCor = GFFromDecimal[2, {0, 1, 1, 3, 1, 0, 2, 0, 1, 1, 3, 2, 0, 1, 0}];
        PolyDecode[K["G(x)"], wordCCor, ListLength → K["k"]] // GFToDecimal

            D(x) = C'(x) / G(x)

Out[ ]= {0, 1, 2, 3, 0, 3, 2, 1, 0}
```

Opět je nejprve vytvořen kód K s pomocí `BCHCode` *se zakázaným dodatečným výpisem, poté opravené slovo c' nad $GF(2^2)$. Prvky jsou zadány převodem z decimálního zápisu. Funkci* `PolyDecode` *je předán generující polynom $G(x)$, který je jedním z parametrů v asociaci reprezentující kód, spolu s opraveným slovem c'. Výstupem je dekódované slovo d nad $GF(2^2)$ zobrazené decimálně. Parametr* `ListLength` *udává jeho požadovanou délku. Pro tu je použita dimenze kódu k jakožto jedna z hodnot v asociaci reprezentující kód. Proběhne výchozí nesystematické dekódování, což dokládá vzorec v dodatečném výpisu.*

4.8 RS kódy

Reed–Solomon kódy jsou nebinární kódy pro opravu volitelného množství chyb. Jedná se o speciální třídu nebinárních BCH kódů popsaných dříve v podkapitole 4.7. Související funkce jsou implementovány v balíčku *RSCode* a ukázky jejich použití se nachází v notebooku `RSKody.nb`.

Nejprve je v sekci 4.8.1 popsána konstrukce kódu, následně v 4.8.2 kódování a na závěr v 4.8.3 oprava a dekódování.

Informace čerpány z [13] a [7, s. 170–177].

4.8.1 Konstrukce kódu

RS kód je dán rozšířeným tělesem $GF((2^{m_1})^{m_2})$ pro $m_2 = 1, m_1 \in \mathbb{N}, m_1 > 1$ a parametrem $\delta \in \mathbb{N}$ udávajícím zaručenou kódovou vzdálenost. Výsledkem je kód délky $n = 2^{m_1} - 1$ se slovy nad $GF(2^{m_1})$ a skutečnou kódovou vzdáleností

$$C_{dist} = \delta\,.$$

Konstrukci RS kódů lze provádět i podle postupu popsaného u nebinárních BCH kódů. Fakt, že se pro sestavení RS kódů používají tělesa s $m_2 = 1$, však

umožňuje některé operace zjednodušit a také provést přesnější odhady některých parametrů výsledného kódu. Například skutečné kódové vzdálenosti, pro niž u obecných BCH kódů máme pouze spodní odhad.

Těleso $GF((2^{m_1})^1)$ je tvořeno polynomy nad $GF(2^{m_1})$ modulo ireducibilní polynom stupně $m_2 = 1$. Obsahuje tedy pouze konstantní polynomy, tedy pouze prvky $GF(2^{m_1})$. Operace sčítání a násobení jsou také totožné jako nad $GF(2^{m_1})$, neboť součinem žádných konstantních polynomů nevznikne polynom stupně 1 nebo vyššího, aby bylo nutné provést redukci modulo ireducibilní definiční polynom. Těleso $GF((2^{m_1})^1)$ proto budeme značit jednoduše jako $GF(2^{m_1})$.

Jedinou situací, kdy je mezi tělesy $GF((2^{m_1})^1)$ a $GF(2^{m_1})$ nutno rozlišovat, je při výpočtu minimálních polynomů. Považujeme-li prvek $\beta \in GF(2^{m_1})$ za prvek rozšířeného tělesa $GF((2^{m_1})^1)$, budeme hledat jeho minimální polynom $M_\beta(x)$ mezi polynomy s koeficienty z $GF(2^{m_1})$, nikoliv ze $\mathbb{Z}_2$. Nad $GF(2^{m_1})$ je minimálním polynomem prvku β jednoduše

$$M_\beta(x) = x + \beta\,.$$

Toho je využito při konstrukci generujcího polynomu $G(x)$. Mějme nějaký primitivní prvek α tělesa $GF(2^{m_1})$ a označme $M_{\#i}(x)$ pro $i \in \mathbb{N}$ minimální polynom prvku α^i považovaného za prvek $GF((2^{m_1})^1)$. Pak pro generující polynom dostáváme postupně

$$\begin{aligned} G(x) &= LCM(M_{\#1}(x), M_{\#2}(x), \ \ldots\ , M_{\#\delta-1}(x))\,, \\ G(x) &= M_{\#1}(x) \cdot M_{\#2}(x) \cdot \ \ldots\ \cdot M_{\#\delta-1}(x)\,, \\ G(x) &= (x+\alpha) \cdot (x+\alpha^2) \cdot \ \ldots\ \cdot (x+\alpha^{\delta-1})\,. \end{aligned}$$

Ze vztahu pro minimální polynom je patrné, že žádné dva různé prvky nemají stejný minimální polynom. Minimální polynomy prvků $\alpha^1, \ldots, \alpha^{\delta-1}$ jsou tak navzájem různé, a tedy i nesoudělné. Jejich nejmenším společným násobkem je jejich součin. Stupeň $G(x)$ je vždy $r = \delta - 1$ a dimenze kódu tím pádem vždy $k = n - r = n - \delta + 1$.

Z redundance kódu $r = \delta - 1$ mimo jiné vyplývá, že kontrolní matice

$$H = \begin{pmatrix} (\alpha^1)^{n-1} & (\alpha^1)^{n-2} & \ldots & (\alpha^1)^2 & \alpha^1 & 1 \\ (\alpha^2)^{n-1} & (\alpha^2)^{n-2} & \ldots & (\alpha^2)^2 & \alpha^2 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ (\alpha^{\delta-1})^{n-1} & (\alpha^{\delta-1})^{n-2} & \ldots & (\alpha^{\delta-1})^2 & \alpha^{\delta-1} & 1 \end{pmatrix}$$

nad $GF(2^{m_1})$ rozměrů $(\delta - 1) \times n$ neobsahuje žádné lineárně závislé řádky. U RS kódů nedochází k žádnému rozepisování prvků matice H do sloupců, jako tomu je u ostatních BCH kódů.

Implementace

Implementace RS kódů je o poznání jednodušší než u obecných nebinárních BCH kódů. Není třeba provádět žádné operace nad rozšířenými tělesy, hledat nejmenší společný násobek minimálních polynomů, ani nijak transformovat matici H.

Konstrukci RS kódů provádí funkce `RSCode` se dvěma povinnými pozičními parametry, obdobně jako v případě funkce `BCHCode`. První parametr specifikuje těleso $GF(2^{m_1})$. Druhý parametr udává zaručenou kódovou vzdálenost $\delta \in \mathbb{N}$. Je-li jako specifikace tělesa zadán ireducibilní polynom nad $\mathbb{Z}_2$ stupně m_1, je tento použit jako definující polynom tělesa $GF(2^{m_1})$. Je-li zadáno celé číslo $m_1 > 1$, bude pro konstrukci $GF(2^{m_1})$ použit výchozí ireducibilní polynom daného stupně.

Prvky $GF(2^{m_1})$ jsou reprezentovány standardně pomocí systémového balíčku *FiniteFields*. Způsob jejich notace v dodatečném výpisu lze ovlivnit třetím pozičním parametrem, předáním některé z funkcí začínajících na `GFTo*` z balíčku *CommonGFCode*.

Primitivní prvek α tělesa $GF(2^{m_1})$ je získán příslušnými funkcemi z balíčku *FiniteFields*. Výpočet minimálních polynomů, generujícího polynomu $G(x)$ a kontrolní matice H je pak velmi jednoduchý. Využívá pouze základní operace nad polynomy a tělesem samotným.

Příklad 1 *RS kód nad $GF(2^3)$ se zaručenou kódovou vzdáleností $\delta = 5$.*

```
In[ ]:= RSCode[3, 5, GFToDecimal]
```

GF(8), δ = 5, n = 7

Minimal polynomials:

i	α^i	$M_{H_i}(x)$
1	2	1x + 2
2	4	1x + 4
3	3	1x + 3
4	6	1x + 6

$G(x) = M_{H1}(x) \cdot M_{H2}(x) \cdot M_{H3}(x) \cdot M_{H4}(x)$

$G(x) = (1x + 2) \cdot (1x + 4) \cdot (1x + 3) \cdot (1x + 6)$

$G(x) = 1x^4 + 3x^3 + 1x^2 + 2x + 3$

k = n − deg[G(x)] = 7 − 4 = 3

$$H = \begin{pmatrix} \alpha^6 & \alpha^5 & \alpha^4 & \alpha^3 & \alpha^2 & \alpha & 1 \\ (\alpha^2)^6 & (\alpha^2)^5 & (\alpha^2)^4 & (\alpha^2)^3 & (\alpha^2)^2 & \alpha^2 & 1 \\ (\alpha^3)^6 & (\alpha^3)^5 & (\alpha^3)^4 & (\alpha^3)^3 & (\alpha^3)^2 & \alpha^3 & 1 \\ (\alpha^4)^6 & (\alpha^4)^5 & (\alpha^4)^4 & (\alpha^4)^3 & (\alpha^4)^2 & \alpha^4 & 1 \end{pmatrix}$$

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

```
Out[ ]= ⟨| type → RSCode, n → 7, k → 3, GF → GF[2, {1, 1, 0, 1}], δ → 5,
  G(x) → x {0, 1, 0}_GF(8) + x^2 {0, 0, 1}_GF(8) + x^4 {0, 0, 1}_GF(8) + {0, 1, 1}_GF(8) + x^3 {0, 1, 1}_GF(8), α → {0, 1, 0}_GF(8),
  H → {{{1, 0, 1}_GF(8), {1, 1, 1}_GF(8), {1, 1, 0}_GF(8), {0, 1, 1}_GF(8), {1, 0, 0}_GF(8), {0, 1, 0}_GF(8), {0, 0, 1}_GF(8)},
    {{1, 1, 1}_GF(8), {0, 1, 1}_GF(8), {0, 1, 0}_GF(8), {1, 0, 1}_GF(8), {1, 1, 0}_GF(8), {1, 0, 0}_GF(8), {0, 0, 1}_GF(8)},
    {{1, 1, 0}_GF(8), {0, 1, 0}_GF(8), {1, 1, 1}_GF(8), {1, 0, 0}_GF(8), {1, 0, 1}_GF(8), {0, 1, 1}_GF(8), {0, 0, 1}_GF(8)},
    {{0, 1, 1}_GF(8), {1, 0, 1}_GF(8), {1, 0, 0}_GF(8), {1, 1, 1}_GF(8), {0, 1, 0}_GF(8), {1, 1, 0}_GF(8), {0, 0, 1}_GF(8)}}, EC → 2 errors |⟩
```

Jako specifikace tělesa $GF(2^3)$ je funkci `RSCode` *předáno pouze celé číslo 3, pro jeho definici bude tudíž použit odpovídající výchozí ireducibilní polynom. Druhým parametrem je zaručená kódová vzdálenost 5. Předáním funkce GFToDecimal je dosaženo decimální notace u prvků $GF(2^3)$ v dodatečném výpisu.*

Výstupem je kód reprezentovaný asociací. Dodatečný výpis dodržuje strukturu použitou u obecných BCH kódů. Je však vynechán nejmenší společný násobek minimálních polynomů při výpočtu $G(x)$ a rovněž jeden krok při sestavování matice H, který demonstroval rozepsání prvků do sloupců.

4.8.2 Kódování

Kódování u RS kódů je prováděno jako u obecných nebinárních BCH kódů a ostatních kódů generovaných polynomem nad $GF(2^{m_1})$. A to tak, že je informační slovo a nad $GF(2^{m_1})$ délky k považováno za seznam koeficientů polynomu $A(x)$, který je vynásoben generujícím polynomem $G(x)$ stupně r. Kódové slovo b nad $GF(2^{m_1})$ délky n je seznam koeficientů polynomu

$$B(x) = A(x) \cdot G(x)\,.$$

Takto provedené kódování není systematické, tedy prvních k slabik b není identické slovu a. Systematického kódování lze dosáhnout předpisem

$$B(x) = A(x) \cdot x^r + (A(x) \cdot x^r)\,\%\,G(x)\,,$$

kde % značí operaci modulo polynom.

Implementace

Jelikož je proces kódování společný všem kódům generovaným polynomem, není realizován samostatnou funkcí v balíčku *RSCode*. Ke kódování je nutné využít balíčku *PolynomialCode*, konkrétně funkce `PolyEncode`, která umí zpracovat jak binární, tak nebinární kódy.

Příklad 2 *Systematické zakódování informačního slova nad $GF(2^3)$ RS kódem se zaručenou kódovou vzdáleností $\delta = 5$.*

```
In[•]:= K = RSCode[3, 5, NoPrint → True];
        wordA = GFFromDecimal[3, {4, 7, 4}];
        PolyEncode[K["G(x)"], wordA, ListLength → K["n"], CodeType → "Systematic"] // GFToDecimal
            B(x) = A(x)·x^r + (A(x)·x^r) % G(x)          where r = deg[G(x)]
Out[•]= {4, 7, 4, 3, 7, 0, 0}
```

Nejprve je definován kód K pomocí funkce `RSCode` *se zakázaným dodatečným výpisem, následně informační slovo a nad tělesem $GF(2^2)$. Prvky jsou zadány v decimální notaci. Funkci* `PolyEncode` *je předán generující polynom*

kódu $G(x)$, který je jedním z klíčů v asociaci reprezentující kód K a slovo a. Výstupem je odpovídající slovo kódové. To je zobrazeno taktéž v decimální notaci. Parametr `ListLength` *udává délku kódového slova, ta je klíčem v asociaci reprezentující kód. Parametrem* `CodeType` *je vynuceno systematické kódování.*

4.8.3 Oprava a dekódování

Oprava RS kódů probíhá ve stejných krocích jako oprava nebinárních BCH kódů. Nikde v ní však nevystupují rozšířená tělesa.

Mějme přijaté slovo c nad $GF(2^{m_1})$ délky n, které lze také chápat jako polynom $C(x)$ nad $GF(2^{m_1})$. Syndromy $s_1, \ldots, s_{\delta-1} \in GF(2^{m_1})$ lze určit buď z kontrolní matice H nebo z polynomu $C(x)$ jako

$$(s_1 \ \ldots \ s_{\delta-1}) = c \cdot H^\top \qquad \text{resp.} \qquad s_i = C(\alpha^i)\,,$$

kde α je primitivní prvek $GF(2^{m_1})$ použitý při konstrukci kódu.

RS kódem se zaručenou kódovou vzdáleností δ lze opravit $\lfloor(\delta-1)/2\rfloor$ chyb. Tato hodnota je proto používána v Peterson–Gorensten–Zierler algoritmu jako horní hranice při hledání nejvyššího ν takového, že matice Θ_ν má nulový determinant.

Matice Θ_ν se skládá z dílčích syndromů, je tedy nad $GF(2^{m_1})$. Stejně jako lokátor $\Lambda(\xi)$, jehož koeficienty jsou na základě Θ_ν vypočteny.

Kořeny lokátoru $\xi_1, \ldots, \xi_\nu$ hledané mezi všemi prvky $GF(2^{m_1})$ udávají pozice chyb $p_1, \ldots, p_\nu \in \{1, \ldots, n\}$. Lze z nich také dle vzorců uvedených u BCH kódů, v nichž figuruje matice Ω, dopočíst tvary chyb $\epsilon_1, \ldots, \epsilon_\nu$ z $GF(2^{m_1})$. S tím rozdílem, že všechny operace probíhají pouze nad $GF(2^{m_1})$. Oprava je poté provedena přičtením ϵ_i k p_i-té slabice slova c, pro všechna $1 \leq i \leq n$.

Postup dekódování opraveného slova c' je stejný jako u obecných nebinárních BCH kódů a ostatních kódů generovanými polynomem $G(x)$ nad $GF(2^{m_1})$. Pro nesystematický kód je dáno předpisem

$$D(x) = C'(x) \,/\, G(x)\,,$$

kde $C'(x)$ je polynom odpovídající slovu c' a seznam koeficientů $D(x)$ je dekódované slovo d nad $GF(2^{m_1})$ délky k. U systematického dekódování je $D(x)$ vypočteno jako

$$D(x) = C'(x) \,/\, x^r\,,$$

kde r je stupeň $G(x)$. V obou případech značí operace / dělení polynomů.

Implementace

Implementace opravy u RS kódů nevyžaduje oproti obecným BCH kódům žádná další rozšíření, je spíše jednodušší. Má ji na starost funkce `RSCorrect` se dvěma povinnými parametry – asociací reprezentující kód a přijatým slovem c, které může být rovněž zadáno jako polynom $C(x)$. Třetím volitelným

parametrem je funkce pro formátování prvků konečného tělesa v dodatečném výpisu.

Příklad 3 *Oprava přijatého slova nad $GF(2^3)$ s chybou na 2. a 4. slabice pomocí RS kódu se zaručenou kódovou vzdáleností $\delta = 5$.*

```
In[•]:= K = RSCode[3, 5, NoPrint → True];
        wordC = GFFromDecimal[3, {4, 0, 4, 1, 7, 0, 0}];
        RSCorrect[K, wordC, GFToDecimal] // GFToDecimal
```

GF(8), δ = 5, n = 7, k = 3

c = {4, 0, 4, 1, 7, 0, 0}

$C(x) = 4x^6 + 4x^4 + 1x^3 + 7x^2$

Syndrome:

i	α^i	$s_i = C(\alpha^i)$
1	2	5
2	4	3
3	3	6
4	6	3

Matrix Θ_v:

Starting from the highest correctable error size v = 2:

$$\Theta_2 = \begin{pmatrix} s_1 & s_2 \\ s_2 & s_3 \end{pmatrix} = \begin{pmatrix} 5 & 3 \\ 3 & 6 \end{pmatrix} \quad \rightarrow \quad \det(\Theta_2) = 6$$

Determinant is non-zero → error size v = 2

Locator polynomial $\Lambda(\xi) = \lambda_2 \xi^2 + \lambda_1 \xi + \lambda_0$:

$$\begin{pmatrix} \lambda_2 \\ \lambda_1 \end{pmatrix} = \Theta_2^{-1} \cdot \begin{pmatrix} s_3 \\ s_4 \end{pmatrix} = \begin{pmatrix} 1 & 5 \\ 5 & 4 \end{pmatrix} \cdot \begin{pmatrix} 6 \\ 3 \end{pmatrix} = \begin{pmatrix} 2 \\ 4 \end{pmatrix}$$

→ $\Lambda(\xi) = 2\xi^2 + 4\xi + 1$

→ roots: $\xi_1 = 4$, $\xi_2 = 6$

Error positions p_1, p_2:

$\xi_1 = 4 = \alpha^2 \quad \rightarrow \quad p_1 = 2$

$\xi_2 = 6 = \alpha^4 \quad \rightarrow \quad p_2 = 4$

Error values ϵ_1, ϵ_2:

$$\Omega = \begin{pmatrix} \xi_1^{-1} & \xi_2^{-1} \\ \xi_1^{-2} & \xi_2^{-2} \end{pmatrix} = \begin{pmatrix} 7 & 3 \\ 3 & 5 \end{pmatrix}$$

$$\begin{pmatrix} \epsilon_1 \\ \epsilon_2 \end{pmatrix} = \Omega^{-1} \cdot \begin{pmatrix} s_1 \\ s_2 \end{pmatrix} = \begin{pmatrix} 3 & 1 \\ 1 & 4 \end{pmatrix} \cdot \begin{pmatrix} 5 \\ 3 \end{pmatrix} = \begin{pmatrix} 7 \\ 2 \end{pmatrix} = \begin{pmatrix} 7 \\ 2 \end{pmatrix}$$

Correction:

c = {4, 0, 4, 1, 7, 0, 0}

e = {0, 7, 0, 2, 0, 0, 0}

c'= {4, 7, 4, 3, 7, 0, 0}

```
Out[•]= {4, 7, 4, 3, 7, 0, 0}
```

Nejprve je definován kód K se zakázaným dodatečným výpisem, následně přijaté slovo c. To je zadáno jako seznam decimálních čísel a následně převedeno na prvky $GF(2^3)$. Výstupem funkce `RSCorrect` *je opravené slovo c', které je následně ještě převedeno do decimální notace. Dodatečný výpis je totožný jako u ostatních nebinárních BCH kódů. Ukazuje všechny fáze opravy, od výpočtu syndromů, přes nalezení lokátoru, až po určení pozic a tvarů chyb a jejich aplikaci na přijaté slovo c. Chyby na pozicích 2 a 4 byly skutečně detekovány a správně opraveny.*

Pro dekódování není v balíčku *RSCode* dedikovaná funkce. Používá se `PolyDecode` z balíčku *PolynomialCode* jakožto protějšek `PolyEncode`. Obě tyto funkce umí pracovat s binárními i nebinárními kódy.

Příklad 4 *Systematické dekódování opraveného slova RS kódu nad $GF(2^3)$ se zaručenou kódovou vzdáleností $\delta = 5$.*

```
In[ ]:= K = RSCode[3, 5, NoPrint → True];
        wordCCor = GFFromDecimal[3, {4, 7, 4, 3, 7, 0, 0}];
        PolyDecode[K["G(x)"], wordCCor, ListLength → K["k"], CodeType → "Systematic"] // GFToDecimal
            D(x) = C'(x) / x^r          where r = deg[G(x)]
Out[ ]= {4, 7, 4}
```

Opět je nejprve vytvořen kód K se zakázaným dodatečným výpisem, poté je zadáno opravené slovo c' pomocí seznamu decimálních čísel, který je následně převeden na seznam prvků $GF(2^3)$. Funkci `PolyDecode` *je zadán generující polynom $G(x)$, který je jedním z parametrů kódu K, spolu se slovem c'. Výstupem je dekódované slovo d nad $GF(2^3)$, které je dodatečně převedeno do decimální notace. Parametr* `ListLength` *udává jeho požadovanou délku. Pro tu je použita dimenze kódu k jakožto jedna z hodnot v asociaci reprezentující kód K. Parametrem* `CodeType` *je vynuceno systematické dekódování.*

Kapitola 5

Bezpečnostní kódy v kryptografii

Tato kapitola shrnuje aktuální stav standardizace post-kvantových kryptosystémů v době psaní práce, tedy k počátku roku 2021. Nejprve je v podkapitole 5.1 popsán dosavadní průběh standardizačního procesu *NIST PQC*. Následně jsou v podkapitole 5.2 stručně charakterizovány tři nejslibnější kryptosystémy založené na bezpečnostních kódech.

5.1 Standardizace post-kvantových kryptosystémů

V posledních desetiletích jsme svědky rozvoje v oblasti výzkumu kvantových počítačů a algoritmů pro ně určených. Ačkoliv konstrukce samotných kvantových počítačů je zatím v plenkách, je sestrojení výkonného kvantového stroje věnována značná pozornost. Jsou pro něj totiž známy efektivní algoritmy k realizaci některých matematických výpočtů, jejichž řešení je na dnešních běžných počítačích obtížné, či neschůdné.

Výkonný kvantový počítač je tudíž příslibem zcela nových výpočetních možností. To ale zároveň představuje velkou výzvu pro současnou kryptografii, která je na neřešitelnosti určitých problémů v rozumné době založena. Zejména se jedná o problémy diskrétního logaritmu a faktorizace celých čísel. Na těch stojí mnohé dnešní asymetrické kryptosystémy jako například RSA, DSA, ECDSA a ECDH. Tyto systémy implementují především algoritmy digitálního podpisu a dohody na klíči. Představují tudíž kritický faktor v zajištění důvěrnosti a autenticity komunikace v rámci Internetu a dalších sítí.

Obavy z prolomení těchto kryptosystémů vedou k rozvoji takzvané post-kvantové kryptografie, anglicky *post-quantum cryptography (PQC)*. Jejím cílem je vyvinout nové kryptografické algoritmy odolné jak vůči kvantovým, tak klasickým počítačům. Ty by měly nahradit stávající asymetrické kryptosystémy. Standardizaci těchto algoritmů se věnuje americký *National Institute of*

Standards and Technology (NIST) ve veřejné soutěži známé pod názvem *NIST PQC Standardization Process*. Hlavním rozcestníkem pro informace týkající se standardizačního procesu jsou webové stránky [17]. Následující informace byly čerpány z průvodních dokumentů této soutěže [18] a [19].

Nejedná se o první veřejnou soutěž vedoucí ke standardizaci nových kryptografických algoritmů, kterou tento institut zaštiťuje. Podobným postupem vznikla také symetrická šifra AES a hešovací funkce SHA-3. Nicméně je očekáváno, že standardizace post-kvantových algoritmů bude v porovnání s těmito dvěma předchozími náročnější. Je to dáno především tím, že schémata asymetrické kryptografie jsou sama o sobě složitější. Dále pak možnosti útoků za využití kvantových počítačů nejsou zdaleka tak dobře prozkoumány jako u klasických počítačů. A v neposlední řadě je očekáván vznik mnoha inovativních postupů založených na nových matematických teoriích, které nejsou současnou kryptografií využívány a jejichž bezpečnost bude nutné důkladně prověřit. Mezi tyto teorie patří vedle mřížek a multivariačních polynomů (polynomů více proměnných) také bezpečnostní kódy.

Každý návrh na standardizaci musí obsahovat popis skupiny algoritmů, které dohromady tvoří jeden kryptosystém, neboli kryptografické schéma, které implementuje jednu či více z následujících funkcionalit:

Asymetrické šifrování *Public-key encryption*

V této kategorii jsou požádovány algoritmy pro generování klíčů, šifrování a dešifrování. Generátor klíčů vrací dvojici soukromého a veřejného klíče tak, aby zprávy zašifrované pomocí veřejného klíče mohly být dešifrovány pouze držitelem klíče soukromého.

Dohoda na klíči *Key encapsulation mechanism (KEM)*

KEM neboli mechanismus pro zapouzdření klíčů je relativně nový termín používaný pro postup stanovení společného symetrického klíče za pomoci asymetrické kryptografie. Požadované jsou algoritmy pro generování klíčů, zapouzdření a odpouzdření. Generátor klíčů vrací stejně jako u předchozí kategorie dvojici soukromého a veřejného klíče. A to takovou, že zapouzdření pomocí veřejného klíče (jediný vstup algoritmu) vrací nový symetrický klíč a šifrový text, ze kterého je držitel soukromého klíče schopen odpouzdřením získat tentýž symetrický klíč.

Tuto dohodu na symetrickém klíči je možné realizovat i za pomoci asymetrického šifrování – držitel veřejného klíče vygeneruje symetrický klíč, zašifruje jej a odešle držiteli soukromého klíče. Přistup KEM však nabízí v porovnání s tímto postupem dvě hlavní výhody. Není nutné řešit generování klíče, tato funkcionalita je zabudována do algoritmu zapouzdření. A zadruhé není nutné řešit padding při šifrování, jenž by byl vzhledem k malé velikosti symetrického klíče pravděpodobně nutný. O vše se starají algoritmy zapouzdření a odpouzdření. V porovnání s asymetrickým

šifrováním nabízí KEM pouze omezené možnosti, to ale znamená také nižší riziko nesprávného použití.

Digitální podpis *Digital signiture*

Stejně jako u předchozích dvou je u digitálního podpisu vyžadován algoritmus pro generování klíčů, dále pak algoritmy podpisu a ověření podpisu. Zpráva podepsaná soukromým klíčem musí být ověřitelná odpovídajícím veřejným klíčem.

Vzhledem k tomu, že funkcionality asymetrického šifrování a dohody na klíči jsou velmi podobné, a dokonce existují bezpečné metody převodu jedné na druhou, došlo prakticky ke sloučení těchto dvou kategorií do jedné. Kandidáti na standardizaci se tak dělí do dvou hlavních skupin: asymetrické šifrování a dohoda na klíči; digitální podpis.

Cílem není vybrat v každé kategorii jediného vítěze. Standardizovaných kryptosystémů může být nakonec několik. Jsou totiž hodnoceny z mnoha hledisek. Lze očekávat, že pro některé aplikace bude některý kryptosystém vhodnější a pro jiné méně. Standardizace několika kryptosystémů založených na různých matematických teoriích je pak výhodná pro případ objevení nedostatků některé z nich.

Primárním hodnotícím hlediskem je samozřejmě bezpečnost. V rámci ní lze dále zohlednit předpokládané prostředky nutné k jejímu prolomení hrubou silou, rizika plynoucí z opakovaného použití téhož klíče, nebo odolnost vůči útokům postranními kanály. Vedle bezpečnosti jsou pak kritickými vlastnostmi rychlost a paměťové nároky jednotlivých algoritmů, velikost klíčů a šifrového textu či pravděpodobnost neúspěšného dešifrování (u některých schémat toto skutečně hrozí i při správné implementaci). Důležitá je též možnost realizace algoritmů na platformách s omezenými prostředky, jejich paralelizace, či začlenění do již existujících protokolů.

Samotný proces standardizace je rozvržen na několik let a členěn do několika kol. V každém kole je zúžen počet kandidátů. Díky tomu může být ve vyšších kolech věnována těm nejslibnějším kandidátům co největší pozornost. Navíc postupující do vyššího kola mají možnost upravit specifikaci svých kryptosystémů, úprava v rámci kola pak již možná není. Výběr postupujících kandidátů provádí NIST na základě pevně stanovených kritérií a komentářů veřejnosti. Důvody ke svému výběru pak uvádí ve volně dostupných zprávách. Ze zpráv po prvním [20] a druhém [21] kole čerpá následující text. Soutěž doprovází i několik konferencí.

Dosavadní vývoj standardizačního procesu a počtu kandidátů je zhruba následující:

Vyhlášení soutěže prosinec 2016

NIST zveřejňuje výzvu k účasti v soutěži. Ta obsahuje jednak požadavky na přihlášení a také pravidla hodnocení. Požadavky zahrnují mimo jiné

kompletní specifikace jednotlivých algoritmů, analýzu bezpečnosti a odolnosti vůči známým útokům, analýzu výpočetní náročnosti jednotlivých operací nebo také dvě kompletní implementace, jednu referenční a jednu optimalizovanou, spolu s testovacími daty.

Celkový počet přihlášených činí 82, pro přijetí do prvního kola je kontrolováno splnění všech požadavků.

První kolo listopad 2017 – leden 2019

Požadavky pro přijetí do soutěže splňuje 48 kryptosystémů v kategorii asymetrického šifrování a dohody na klíči. Z nich 19 je založeno na bezpečnostních kódech a podobné množství na mřížkách, dohromady tak představují drtivou většinu v této kategorii.

V kategorii digitálního podpisu je do soutěže přijato 19 kryptosystémů. Z nich pouze 3 jsou založeny na bezpečnostních kódech. Většinu tvoří mřížkové systémy v kombinaci s multivariačními.

Tyto kryptosystémy jsou podrobeny základní analýze ze strany pracovníků institutu a veřejnosti, ti nejslibnější jsou vybráni pro postup do druhého kola.

Druhé kolo leden 2019 – červenec 2020

U asymetrického šifrování a dohody na klíči dochází ke snížení počtu kandidátů na 17 (ze 48). Zastoupení matematických teorií se příliš nemění, dominují bezpečnostní kódy a mřížky. Kódy jsou zastoupeny v počtu 7, konkrétně jde například o kódy Goppa, BCH *(Bose–Chaudhuri–Hocquenghem)*, LDPC *(Low-Density Parity-Check)*, MDPC *(Moderate-Density Parity-Check)*, RS *(Reed–Solomon)*, RM *(Reed–Muller)*, nebo LRPC *(Low-Rank Parity-Check)*.

V kategorii digitálního podpisu je snížen počet kandidátů na 9 (z 19). Bezpečnostní kódy se však neukázaly být vhodným nástrojem pro implementaci této funkcionality, do druhého kola nepostoupil žádný na nich založený kryptosystém.

V rámci druhého kola jsou kandidáti podrobeni detailnější analýze než v prvním kole, samotný postup je však velmi podobný.

Třetí kolo červenec 2020 –

Do třetího kola vstupují někteří kandidáti jako finalisté a někteří jako náhradníci. U finalistů je očekáván hotový návrh na standardizaci na konci tohoto kola (zřejmě do konce roku 2021). U náhradníků není standardizace vyloučena, vyžadují však další čas na analýzu, či optimalizaci. Bude pro ně pravděpodobně otevřeno kole čtvrté.

Mezi 7 finalisty dominují kryptosystémy založené na mřížkách. Ty byly zároveň označeny za nejslibnější pro obecné použití, a to v obou kategoriích. U asymetrického šifrování a digitálního podpisu se však mezi ně

dostal i systém *Classic McEliece* založený na kódech. Dominanci mřížkových systémů mezi schématy na digitální podpis narušuje jeden multivariační.

Mezi 8 náhradníky stále (v kategorii asymetrického šifrování a dohody na klíči) figurují 2 systémy postavené na bezpečnostních kódech – *BIKE* a *HQC*.

5.2 Představení kandidátů založených na kódech

V předchozí podkapitole byl popsán dosavadní průběh standardizace postkvantových kryptosystémů v rámci *NIST PQC*. Do třetího kola postoupily tři kryptosystémy založené na bezpečnostních kódech. Z toho systém *Classic McEliece* jako finalista a systémy *BIKE* (*Bit Flipping Key Encapsulation*) a *HQC* (*Hamming Quasi-Cyclic*) jako náhradníci. Tato kapitola nabízí jejich základní popis a srovnání založené převážně na zprávách z průběhu soutěže [20] a [21]. Některé informace byly čerpány také ze samotných specifikací daných kryptosystémů [22], [23] a [24], tak jak byly do soutěže zařazeny.

Ve snaze porovnat jednotlivé kandidáty byly vytvořeny dvě tabulky s parametry jako je velikost veřejných a soukromých klíčů nebo rychlost generování klíčů, šifrování a dešifrování. Jako zdroje dat byly použity benchmark asymetrických kryptosystémů *eBATS* [25] a portál *PQC WIKI* [26] odkazované ve zprávě [21]. Uvedené hodnoty je však třeba brát pouze jako orientační. Neboť jen zřídkakdy nastává mezi všemi zdroji shoda. To je dáno pravděpodobně různými formáty pro uložení klíčů a v případě rychlosti algoritmů samozřejmě také jejich konkrétní implementací a způsobem měření. Tyto údaje by proto měly být v budoucnu podrobeny detailnější analýze.

Systém *Classic McEliece* je založen na kryptosystému McEliece z roku 1978, který jako první představil myšlenku využití bezpečnostních kódů v asymetrické kryptografii. Ačkoliv ve svém názvu nese slůvko *Classic*, obsahuje oproti původnímu návrhu mnohé úpravy vedoucí k navýšení efektivity i bezpečnosti. Stejně jako původní verze však využívá binárních Goppa kódů.

Problém, na němž je kryptosystém postaven, spočívá ve složitosti dekódování kódového slova s chybou za znalosti pouze generující matice. Je-li G generující matice lineárního (n, k)-kódu schopného opravit až t chyb, pak dekódování slova c bez chyby lze provést řešením soustavy $xG = c$. Pokud však c obsahuje chybu, tato soustava řešení nemá. Při znalosti kódu by šlo slovo c nejprve opravit, předpokládáme však, že matice G vypadá náhodně a žádnou informaci o použitém kódu neprozrazuje. Zbývá tak jedině zkoušet všechna kódová slova (těch je 2^k) a hledat takové, jehož vzdálenost od c je menší než t. Alternativně zkoušet odhadnout, kterých t z n bitů slova c je chybných. Obě tyto metody jsou pro velká n, k a t neschůdné.

Pro představu, v základní verzi kryptosystému, jejíž prolomení by mělo být minimálně stejně tak náročné jako prolomení symetrické šifry AES-128,

jsou parametry Goppa kódu $n = 3488$, $k = 2720$ a $t = 64$. Výhodou Goppa kódů je to, že velké množství z nich má stejné parametry, ale odlišná kódová slova. Jejich generující matice je pak nerozlišitelná od náhodného lineárního kódu.

Mezi silné stránky kryptosystému patří dlouholetá historie jeho studování, a tedy poměrně velká důvěra v jeho bezpečnost. Co však nemluví v jeho prospěch jsou velikosti soukromých a veřejných klíčů. Ty násobně (u veřejného klíče dokonce mnohonásobně) převyšují velikosti klíčů konkurenčních mřížkových systémů. Pro srovnání v tabulce jsou vybrány takové verze kryptosystémů, jenž nabízí přibližně stejnou míru zabezpečení – náročnost jejich prolomení nesmí být nižší než u šifry AES-128.

Kryptosystém	VK	SK	Gen. klíčů	Zapouzd.	Odpouzd.
Classic McEliece	261120	6452	206369	38	543
CRYSTALS-KYBER	800	1632	21	35	29
NTRU	699	935	136	19	32
SABER	672	1568	63	87	78

Tabulka 5.1: Orientační srovnání systému *Classic McEliece* s mřižkovými systémy. Velikosti veřejných a soukromých klíčů jsou v bytech. Rychlosti algoritmů v 1000 cyklech na procesoru Intel Core i5-1030NG7. [25]

Je-li součástí některého z klíčů generující či kontrolní matice kódu velkých rozměrů, hrozí, že tato skutečnost bude mít negativní vliv i na velikost samotných klíčů, jako jsme toho svědky u systému *Classic McEliece*. Jednou z metod, jak v takovém případně dosáhnout snížení velikosti klíče, je ukládání matice v systematickém tvaru. Jednotkovou podmatici lze při uložení vynechat. Na druhou stranu je pak nutné zajistit, aby tímto krokem nedošlo k narušení bezpečnosti systému.

Druhým možným přístupem ke zmenšení velikosti klíče je využití cyklických a kvazi-cyklických kódů. U cyklických kódů platí, že libovolný cyklický posuv kódového slova je opět kódové slovo. U kvazi-cyklických kódů je dán parametr $r \in \mathbb{N}$ a musí platit, že libovolný cyklický posuv kódového slova o r míst je opět kódovým slovem. U takovýchto kódů je možné sestrojit cyklické generující a kontrolní matice, tedy takové, jejichž každý řádek je cyklickým posuvem řádku prvního. Místo celé matice tak stačí do klíče uložit pouze jeden její řádek.

Na kvazi-cyklických kódech jsou založeny kryptosystémy *BIKE* a *HQC*. Ty byly vybrány do třetího kola standardizace jako náhradníci.

Systém *BIKE* je založen na kvazi-cyklických MDPC *(Moderate-Density Parity-Check)* kódech. Systém *HQC* nabízí dvě možné kombinace kódů. Původní verze je postavena na součinu BCH kódu s kódem opakovacím. V průběhu soutěže byla navržena verze stojící na konkatenaci RS a RM kódu. Ta

dosahuje lepších výsledků vzhledem k velikosti klíčů.

Velikosti klíčů u obou těchto systémů dosahují v porovnání s kryptosystémem *Classic McEliece* výrazně nižších hodnot. Představují tak vhodná schémata k obecnému použití. Za mřížkovými systémy však stále zaostávají. U obou může navíc v ojedinělých případech dojít k selhání opravy kódového slova, a tedy i selhání příslušného dešifrovacího, či odpouzdřovacího algoritmu. Zatímco *HQC* nabízí precizní důkaz své bezpečnosti a toho, že pravděpodobnost selhání opravy je pod stanovenou mezí, u systému *BIKE* nebyla ještě míra pravděpodobnosti selhání pevně stanovena. V současné době tak lze *BIKE* doporučit pouze pro aplikace, jež jeden pár klíčů použijí k šifrování nejvýše jednou. Naopak výhodou systému *BIKE* oproti *HQC* jsou menší klíče.

Tabulka nabízí srovnání obou systémů opět v základních konfiguracích odpovídajících bezpečnosti šifry AES-128.

Kryptosystém	VK	SK	Gen. klíčů	Zapouzd.	Odpouzd.
BIKE	1541	1821	430	107	978
HQC-BCH	3024	3064	259	402	680
HQC-RMRS	2607	2647	215	358	660

Tabulka 5.2: Orientační srovnání systémů *BIKE* a *HQC*. Velikosti veřejných a soukromých klíčů jsou v bytech. Rychlosti algoritmů v 1000 cyklech na procesoru Intel Core i5-1030NG7. [25]

Závěr

V této práci byla vytvořena sada výukových materiálů, která pokrývá všechny pokročilé bezpečnostní kódy specifikované v zadání. Tato sada je kompatibilní s materiály z předchozí bakalářské práce, což ve výsledku znamená více než deset jednotně zpracovaných typů bezpečnostních kódů. Sada je v souladu se zadáním tvořena knihovnou funkcí v systému Wolfram Mathematica a ukázkovými notebooky demonstrujícími jejich použití. Knihovna prošla testováním a dle dosavadních výsledků pracuje správně.

Všechny vytvořené funkce přehledně vypisují dodatečné informace o průběhu daného výpočtu. Tento výpis tvoří převážně matematické výrazy. Jejich význam je pak detailněji popsán a vysvětlen v samotném textu práce. Cenou za tvorbu dodatečného výpisu je zvýšená složitost jednotlivých funkcí, což má negativní dopad na jejich výkonnost a čitelnost.

Nově vytvořené výukové materiály dodržují formát těch starých. Shoda ale není stoprocentní. Byl například vyvinut nový způsob reprezentace bezpečnostních kódů nebo zvolena mírně odlišná organizace ukázkových notebooků. Tyto změny však nebyly aplikovány na již existující materiály.

V závěru práce byl popsán aktuální vývoj standardizace post-kvantových kryptosystémů s důrazem na uplatnění právě bezpečnostních kódů. Ačkoliv se standardizace kryptosystému založeného na kódech jeví jako reálná, alternativní systémy založené na mřížkách dosahují lepších parametrů, a byly proto označeny za slibnější kandidáty pro obecné použití.

Seznam použité literatury

1. KOLENÍK, Stanislav. *Podpora výuky bezpečnostních kódů v programu Wolfram Mathematica*. Praha, 2019. Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií. [Kopie souboru uložena na přiloženém CD].
2. MYSLIVEC, Vojtěch. *Asymetrický šifrovací algoritmus McEliece*. Praha, 2016. Dostupné také z: `https://github.com/VojtechMyslivec/mceliece-mathematica`. Diplomová práce. České vysoké učení technické v Praze, Fakulta informačních technologií. [Kopie souboru uložena na přiloženém CD].
3. PANARIO, Daniel; VIOLA, Alfredo. Analysis of Rabin's Polynomial Irreducability Test. In: *Proc. of the 3rd Latin American Symposium on Theoretical Informatics (LATIN'98)*. Campinas, Brazil, 1998, s. 1–10. Dostupné také z: `https://www.colibri.udelar.edu.uy/jspui/bitstream/20.500.12008/3455/1/TR0116.pdf`.
4. PLUHÁČEK, Alois. *Materiály k předmětu NI-BKO (Bezpečnostní kódy) na FIT ČVUT v Praze: Kódy pro opravu shluku chyb* [online]. 2021 [cit. 2021-04-15]. Dostupné z: `https://courses.fit.cvut.cz/MI-BKO/media/lectures/BKO-1-05-E-SH1.pdf` [Soubor přístupný po přihlášení do sítě ČVUT – kopie souboru uložena na přiloženém CD].
5. PLUHÁČEK, Alois. *Materiály k předmětu NI-BKO (Bezpečnostní kódy) na FIT ČVUT v Praze: Oprava shluků chyb* [online]. 2021 [cit. 2021-04-15]. Dostupné z: `https://courses.fit.cvut.cz/MI-BKO/media/lectures/BKO-1-06-F-SH2.pdf` [Soubor přístupný po přihlášení do sítě ČVUT – kopie souboru uložena na přiloženém CD].
6. PLUHÁČEK, Alois. *Materiály k předmětu NI-BKO (Bezpečnostní kódy) na FIT ČVUT v Praze: Oprava shluků chyb* [online]. 2019 [cit. 2021-04-15]. Dostupné z: `https://courses.fit.cvut.cz/NI-BKO/@B192/`

media/lectures/BKO-1-06-F-SH2.pdf [Soubor přístupný po přihlášení do sítě ČVUT – kopie souboru uložena na přiloženém CD].

7. LIN, Shu; COSTELLO, Daniel J. *Error Control Coding: Fundamentals and applications* [online]. Englewood Cliffs: Prentice-Hall, Inc., 1983 [cit. 2021-04-15]. ISBN 0-13-283796-X. Dostupné z: http://index-of.co.uk/Information-Theory/Error%20Control%20Coding%20Fundamentals%20and%20Applications%20-%20Shu%20Lin.pdf.
8. PLUHÁČEK, Alois. *Materiály k předmětu NI-BKO (Bezpečnostní kódy) na FIT ČVUT v Praze: Součty a součiny kódů a kódy RM* [online]. 2021 [cit. 2021-04-15]. Dostupné z: https://courses.fit.cvut.cz/MI-BKO/media/lectures/BKO-1-07-G-ME.pdf [Soubor přístupný po přihlášení do sítě ČVUT – kopie souboru uložena na přiloženém CD].
9. ADÁMEK, Jiří. *Kódování*. Praha: SNTL – Nakladatelství technické literatury, 1989.
10. ENGELBERT, Daniela; OVERBECK, Raphael; SCHMIDT, Arthur. A Summary of McEliece-Type Cryptosystems and their Security. *Journal of Mathematical Cryptology*. 2006. Dostupné také z: https://doi.org/10.1515/JMC.2007.009.
11. BERLEKAMP, Elwyn R. Goppa codes. *IEEE Transactions on Information Theory*. 1973, roč. 19, č. 5, s. 590–592. Dostupné také z: https://doi.org/10.1109/TIT.1973.1055088.
12. PLUHÁČEK, Alois. *Materiály k předmětu NI-BKO (Bezpečnostní kódy) na FIT ČVUT v Praze: Dvojkové kódy BCH* [online]. 2021 [cit. 2021-04-15]. Dostupné z: https://courses.fit.cvut.cz/MI-BKO/media/lectures/BKO-1-08-H-BCH.pdf [Soubor přístupný po přihlášení do sítě ČVUT – kopie souboru uložena na přiloženém CD].
13. PLUHÁČEK, Alois. *Materiály k předmětu NI-BKO (Bezpečnostní kódy) na FIT ČVUT v Praze: Kódy pro opravy slabik, kódy RS* [online]. 2021 [cit. 2021-04-15]. Dostupné z: https://courses.fit.cvut.cz/MI-BKO/media/lectures/BKO-1-09-I-RS.pdf [Soubor přístupný po přihlášení do sítě ČVUT – kopie souboru uložena na přiloženém CD].
14. WOLFRAM. Finite Fields Package. *Documentation center* [online]. 2021 [cit. 2021-04-20]. Dostupné z: https://reference.wolfram.com/language/FiniteFields/tutorial/FiniteFields.html.
15. PLUHÁČEK, Alois. *Materiály k předmětu NI-BKO (Bezpečnostní kódy) na FIT ČVUT v Praze: Konvoluční kódy a turbo kódy* [online]. 2021 [cit. 2021-04-15]. Dostupné z: https://courses.fit.cvut.cz/MI-BKO/media/lectures/BKO-1-10-J-KKT.pdf [Soubor přístupný po přihlášení do sítě ČVUT – kopie souboru uložena na přiloženém CD].

16. STAROSTA, Štěpán. *Materiály k předmětu NI-MPI (Matematika pro informatiku) na FIT ČVUT v Praze: Přednáška 2, Algebra II* [online]. 2021 [cit. 2021-04-20]. Dostupné z: `https://courses.fit.cvut.cz/MI-MPI/latex/lectures/czech/mi-mpi-prednaska-02-algebra-II-slides.pdf` [Soubor přístupný po přihlášení do sítě ČVUT – kopie souboru uložena na přiloženém CD].

17. NIST. *Post-Quantum Cryptography* [online]. 2021 [cit. 2021-05-01]. Dostupné z: `https://csrc.nist.gov/projects/post-quantum-cryptography`.

18. NIST. *Report on Post-Quantum Cryptography: NISTIR 8105* [online]. 2016 [cit. 2021-05-01]. Dostupné z: `http://dx.doi.org/10.6028/NIST.IR.8105`.

19. NIST. *Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process* [online]. 2016 [cit. 2021-05-01]. Dostupné z: `https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf`.

20. NIST. *Status Report on the First Round of the NIST Post-Quantum Cryptography Standardization Process: NISTIR 8240* [online]. 2019 [cit. 2021-05-01]. Dostupné z: `https://doi.org/10.6028/NIST.IR.8240`.

21. NIST. *Status Report on the Second Round of the NIST Post-Quantum Cryptography Standardization Process: NISTIR 8309* [online]. 2020 [cit. 2021-05-01]. Dostupné z: `https://doi.org/10.6028/NIST.IR.8309`.

22. MCELIECE, Classic. *Classic McEliece: conservative code-based cryptography* [online]. 2020 [cit. 2021-05-01]. Dostupné z: `https://classic.mceliece.org/nist/mceliece-20201010.pdf`.

23. BIKE. *BIKE: Bit Flipping Key Encapsulation* [online]. 2020 [cit. 2021-05-01]. Dostupné z: `https://bikesuite.org/files/v4.1/BIKE_Spec.2020.10.22.1.pdf`.

24. HQC. *HammingQuasi-Cyclic (HQC)* [online]. 2020 [cit. 2021-05-01]. Dostupné z: `http://pqc-hqc.org/doc/hqc-specification_2020-10-01.pdf`.

25. BERNSTEIN, Daniel J.; LANGE, Tanja. *eBACS: ECRYPT Benchmarking of Cryptographic Systems* [online]. 2021 [cit. 2021-05-01]. Dostupné z: `https://bench.cr.yp.to/results-kem.html`.

26. FLORIDA ATLANTIC UNIVERSITY. *PQC WIKI: A Platform for NIST Post-Quantum Cryptography Standardization* [online]. 2021 [cit. 2021-05-01]. Dostupné z: `https://pqc-wiki.fau.edu/w/Special:DatabaseHome`.

Příloha **A**

Obsah přiloženého CD

```
/
├── Text/ ..... text práce
│   ├── DP_Kolenik_Stanislav_2021.pdf ..... práce ve formátu PDF
│   └── Latex/ ..... práce ve formátu LaTeX
│
├── Reseni/ ..... hlavní výstup praktické části
│   ├── Packages/ ..... knihovna
│   │   ├── *.wl ..... balíčky
│   │   └── Stylesheet.nb ..... definice stylů
│   └── *.nb ..... ukázkové notebooky
│
├── Testovani/ ..... testování kódu
├── Pomocne/ ..... pomocné soubory
├── Literatura/ ..... literatura dostupná ze sítě ČVUT
└── Readme.txt ..... stručný popis obsahu CD
```