

Přenos dat a počítačové sítě

Projekt 2

Obsah

Zabezpečovací linkové kódy pro přenos dat - Lineární kód	1
Zadání	1
Použité prostředky.....	1
Forma odevzdání.....	1
Literatura	1
Vlastnosti zadané matice.....	2
Úpravy na systematický kód.....	2
Výpočet kontrolní matice.....	2
Experimentální ověření vlastností.....	3
Výsledek	3
Implementace	3
Matice.....	3
Kódování.....	4
Detekce a opravy	4
Testování	4
Přílohy	5
Význam syndromu	5
Literatura.....	6

Zabezpečovací linkové kódy pro přenos dat - Lineární kód

Zadání

Cílem projektu je vyzkoušet si kódování, dekódování, a opravu dat zabezpečených lineárním linkovým kódem.

Zabezpečte data pomocí lineárního kódu generovaného maticí. Určete kontrolní matici. Při dekódování správně detekujte a opravte chyby. Popište princip algoritmu, vypočtenou kontrolní matici a demonstруйте funkčnost na příkladu (dávka *testuj*).

Generující matice kódu (6,3):

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

Úkolem je vytvořit program se jménem *kodovani*, který načte vstupní data (ze standardního vstupu) a podle zadaných parametrů je zakóduje nebo dekóduje s případnou opravou. Výsledek vytiskne na standardní výstup. Syntaxe spouštění programu:

```
kodovani (-c | -d)
```

kde -c znamená kódování a -d dekódování. Příklad použití:

```
kodovani -c < data.txt # kódování  
kodovani -d < kod.txt # dekódování
```

Vstupní data jsou tvořena posloupností libovolných binárních dat (jako testovací data použijte například *jpg*, *pdf*, *doc*, *txt*, *bmp* apod.). Jestliže vstupní data netvoří potřebný počet bitů, doplňte zbývající bity bloku nulovými bity. Doplňené bity nezapomeňte při dekódování odstraňovat.

Součástí odevzdaného projektu bude dávka *testuj*, která bude demonstrovat funkčnost vašeho řešení na vámi zvoleném vstupním souboru dat (kódování, dekódování).

Použité prostředky

Program vytvořte v jazyce C/C++ tak, aby byl přeložitelný na serveru EVA (eva.fit.vutbr.cz) nebo na SUNech (např. sun00.fit.vutbr.cz). V dokumentaci uveďte na kterém ze serverů je projekt přeložitelný a funkční.

Pozor: Nepřeložitelné projekty nebudou hodnoceny.

Forma odevzdání

- zip archiv se jménem *login.zip* (např. *xnovak00.zip*).
- součástí archívu budou:
 - funkční Makefile
 - komentované zdrojové kódy programů (*kodovani.c*)
 - dokumentace v pdf, ps nebo ASCII - v češtině (*dokumentace.pdf*, *dokumentace.ps*, *dokumentace.txt*)
 - testovací skript *testuj*, který projekt přeloží a otestuje jeho funkčnost na příkladu
 - soubor testovacích dat pro dávku *testuj*

Dodržujte zadaná jména souborů! Dodržujte parametry spouštění projektu - implementujte projekt tak, aby jej bylo možné spouštět z dávkových souborů (žádné interakce s uživatelem).

Literatura

- přednášky z PDT
- poznámky ze cvičení PDT
- doplňkové texty ke cvičením

Vlastnosti zadané matice

Zadaná matice kódu 3,6 zakóduje tři bity vstupních dat na šest výstupních bitů. Obsah matice je konkrétně:

$$G_{ZADANI} = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \quad (1)$$

Protože tato matice není v systematickém tvaru, tak ji na tento tvar převedeme. K tomuto převodu jsou dovoleny používat sloupcové a řádkové úpravy jako jsou úplná výměna nebo součet (v binárním tvaru se použije bitové XOR).

Úpravy na systematický kód

První úprava je přičtení prvního řádku k poslednímu:

$$G_{1+3} = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \quad (2)$$

Další je pak výměna prvního sloupce s pátým:

$$G_{systematická} = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{pmatrix} \quad (3)$$

Ted' máme kódovou matici v systematickém tvaru, který je obecně zapsatelný jako

$$G_{k,n} = (I_{k,k} \mid P_{k,r}) \quad (4)$$

A v našem případě je rozdělení zde:

$$G_{systematická} = \left(\begin{array}{ccc|ccc} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{array} \right) \quad (5)$$

Výpočet kontrolní matice

Pokud máme systematickou matici, tak pak kontrolní matici lze sestavit jako:

$$H_{r,n} = (P_{r,k}^T \mid I_{r,r}) \quad (6)$$

Po provedení transpozice části P, ještě můžeme provést transpozici celé matice H, ať je názornější co je vstupem a co výstupem (vstup vždy přichází zleva a výstup dostaneme dolů):

$$H = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad (7)$$

Experimentální ověření vlastností

Protože jsem si nebyl jist schopnostmi detekce, udělal jsem experimentální výpočty. Můj experiment probíhal následovně:

- vygeneroval jsem všechny vstupy (0-7) a tyto se zakódovali - dostal jsem 8 bezchybných výsledků:

```
[000] => 000000  
[001] => 001011  
[010] => 010101  
[011] => 011110  
[100] => 100100  
[101] => 101111  
[110] => 110001  
[111] => 111010
```

- výsledky bez chyb jsem modifikoval vždy o jeden bit, a takto jsem dostal z jednoho správného výsledku, dalších 6 - dle toho který bit se modifikoval
- uvedené relace dvojic jsem zanesl do tabulky
- tabulku jsem seřadil dle výsledku, ať ho lze jednoduše identifikovat (viz přílohu)

Výsledek

Výsledné zjištění bylo zajímavé, protože jsem zjistil následovní vlastnosti:

- opravit dokážeme určitou část jednobitových chyb:
 - chybu na 2., 3., 5., 6. bitu ANO
 - chybu na 1. nebo 4. bitu - NE
- detekovat můžeme malou část dvoubitových chyb
 - to jsou ty co vyvolají syndrom 6 nebo 7
 - zbytek - většina - jsou takové, že vyvolají jiné syndromy nebo dokonce pozmění přenesená data tak, že dostaneme syndrom 0 (bezchybnost)

Slabost zabezpečovacích schopností kódu ale vyplývá z návrhu matice - kódovací část, P, není symetrická (nevytvoří úplné kombinace). Kdyby se matice z (5) upravila na následovní tvar změnou vyznačeného bitu, byla by schopnost detekovat 1 bitovou chybu stoprocentní:

$$G_{OK} = \left(\begin{array}{ccc|ccc} 1 & 0 & 0 & 1 & \underline{1} & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{array} \right) \quad (8)$$

Implementace

Projekt byl bez problému otestován pomocí dávky *testuj* na serveru *eva.fit.vutbr.cz* a dávka *testuj* je obdobná jako v prvním projektu.

Matice

Matice jsem implementoval pomocí polí, kde byly jednotlivé bity již zkonvertovány do jedné hodnoty - takže se jedná o efektivní způsob:

```
// kodovaci tabulka  
int G[3] = { 0x24 // 100100  
           , 0x15 // 010101  
           , 0x0B // 001011  
           };  
  
// dekodovaci tabulka  
int H[6] = { 0x04 // 100  
           , 0x05 // 101  
           , 0x03 // 011  
           , 0x04 // 100  
           , 0x02 // 010  
           , 0x01 // 001  
           };
```

Kódování

Proces kódování je jednoduchý výpočet sestavující blok výstupních bitů pomocí vstupních bitů a kódovací matice G. Protože výstup není zarovnan na 8 bitů, je někdy potřeba doplnit poslední bajt nulovými bity aby byl kompletní a teda zapsán. Tuto operaci provádí procedura *splachni* (obdoba anglického *flush*).

```
void koduj() {
    // pokud je vstup
    while(cti(3)) {
        // provadime kodovani
        c = 0;
        if ( m & 0x04 ) c = c ^ G[0];
        if ( m & 0x02 ) c = c ^ G[1];
        if ( m & 0x01 ) c = c ^ G[2];
        // zapiseme zakodovane slovo
        zapis(6);
    }
    splachni();
}
```

Detekce a opravy

Dekódování používá stejného principu s rozdílem že nedostáváme po vynásobení maticí data, ale syndrom chyby dle kterého se pak provede patřičná akce.

Někdy je třeba opravit bit (*BIT1-BIT6* konstanty), někdy je výsledek správný (*CORRECT*) a někdy jde o kolizi opravy - nevíme který bit se má opravit (*COLLISION*) případně o vícebitovou chybu (*MASS_ERROR*).

```
// specialni akce
#define CORRECT          0x00
#define BIT1             0x20
#define BIT2             0x10
#define BIT3             0x08
#define BIT4             0x04
#define BIT5             0x02
#define BIT6             0x01
#define COLLISION        0xFE
#define MASS_ERROR       0xFF
```

Speciálním případem je chyba jenom v kontrolním kódu - ty dva stavy kde k tomuto dochází jsou u mě označeny jako *CORRECT*, protože oprava se netýká datových bitů. Tato vlastnost vznikla použitím systematického kódu, resp. převedením zadání na systematický kód.

```
// co se syndromem
int akce[8] = { /* syndrom = 0 */ CORRECT
               , /* syndrom = 1 */ CORRECT // BIT6
               , /* syndrom = 2 */ CORRECT // BIT5
               , /* syndrom = 3 */ BIT3
               , /* syndrom = 4 */ COLLISION
               , /* syndrom = 5 */ BIT2
               , /* syndrom = 6 */ MASS_ERROR
               , /* syndrom = 7 */ MASS_ERROR
               };
```

Testování

Pro testovací účely jsem ještě udělal program *ruseni* který modifikuje ty bity, které umí program detekovat. Jiné rušení modifikující data k nepoznání jsem provedl doplněním textu k zašifrovanému souboru (dosahuje se teda 2bitových chyb - syndromu 6, 7 nebo kolizi, syndrom 4).

```
#define BIT2             0x10
#define BIT3             0x08
#define BIT5             0x02
#define BIT6             0x01

int r[4] = { BIT2, BIT3, BIT5, BIT6 };

while(cti(6)) { c = m ^ r[ i++ % 4 ]; zapis(6); }
```

Přílohy

Význam syndromu

```
[000000] => syndrom = 0, CORRECT as 000
[001011] => syndrom = 0, CORRECT as 001
[010101] => syndrom = 0, CORRECT as 010
[011110] => syndrom = 0, CORRECT as 011
[100100] => syndrom = 0, CORRECT as 100
[101111] => syndrom = 0, CORRECT as 101
[110001] => syndrom = 0, CORRECT as 110
[111010] => syndrom = 0, CORRECT as 111
[000001] => syndrom = 1, ERROR 6 in 000
[001010] => syndrom = 1, ERROR 6 in 001
[010100] => syndrom = 1, ERROR 6 in 010
[011111] => syndrom = 1, ERROR 6 in 011
[100101] => syndrom = 1, ERROR 6 in 100
[101110] => syndrom = 1, ERROR 6 in 101
[110000] => syndrom = 1, ERROR 6 in 110
[111011] => syndrom = 1, ERROR 6 in 111
[000010] => syndrom = 2, ERROR 5 in 000
[001001] => syndrom = 2, ERROR 5 in 001
[010111] => syndrom = 2, ERROR 5 in 010
[011100] => syndrom = 2, ERROR 5 in 011
[100110] => syndrom = 2, ERROR 5 in 100
[101101] => syndrom = 2, ERROR 5 in 101
[110011] => syndrom = 2, ERROR 5 in 110
[111000] => syndrom = 2, ERROR 5 in 111
[001000] => syndrom = 3, ERROR 3 in 000
[000011] => syndrom = 3, ERROR 3 in 001
[011101] => syndrom = 3, ERROR 3 in 010
[010110] => syndrom = 3, ERROR 3 in 011
[101100] => syndrom = 3, ERROR 3 in 100
[100111] => syndrom = 3, ERROR 3 in 101
[111001] => syndrom = 3, ERROR 3 in 110
[110010] => syndrom = 3, ERROR 3 in 111
[100000] => syndrom = 4, ERROR 1 in 000, ERROR 4 in 100
[101011] => syndrom = 4, ERROR 1 in 001, ERROR 4 in 101
[110101] => syndrom = 4, ERROR 1 in 010, ERROR 4 in 110
[111110] => syndrom = 4, ERROR 1 in 011, ERROR 4 in 111
[000100] => syndrom = 4, ERROR 4 in 000, ERROR 1 in 100
[001111] => syndrom = 4, ERROR 4 in 001, ERROR 1 in 101
[010001] => syndrom = 4, ERROR 4 in 010, ERROR 1 in 110
[011010] => syndrom = 4, ERROR 4 in 011, ERROR 1 in 111
[010000] => syndrom = 5, ERROR 2 in 000
[011011] => syndrom = 5, ERROR 2 in 001
[000101] => syndrom = 5, ERROR 2 in 010
[001110] => syndrom = 5, ERROR 2 in 011
[110100] => syndrom = 5, ERROR 2 in 100
[111111] => syndrom = 5, ERROR 2 in 101
[100001] => syndrom = 5, ERROR 2 in 110
[101010] => syndrom = 5, ERROR 2 in 111
[000110] => syndrom = 6
[111100] => syndrom = 6
[011000] => syndrom = 6
[010011] => syndrom = 6
[101001] => syndrom = 6
[110111] => syndrom = 6
[001101] => syndrom = 6
[100010] => syndrom = 6
[111101] => syndrom = 7
[101000] => syndrom = 7
[001100] => syndrom = 7
[010010] => syndrom = 7
[100011] => syndrom = 7
[000111] => syndrom = 7
[110110] => syndrom = 7
[011001] => syndrom = 7
```

Literatura

- [DK] **Datová komunikace - Úvod do teorie informací a kódování**
Prof. Ing. Dalibor Bielek, CSc., Ústav telekomunikací
Fakulta elektrotechniky a komunikačních technologií - VUT Brno
<http://user.unob.cz/biolek/vyukaVUT/skripta/DKO.pdf>
- [C] **Jazyk C - přednášky**
Dr. Ing. Petr Peringer
<http://www.fit.vutbr.cz/study/courses/IJC/public/Prednasky/IJC.pdf>
- [BASH] **bash - GNU Bourne-Again SHell**
manuálová stránka programu bash(1), man bash