Αριστοτέλειο Πανεπιστήμιο Θεσσαλονίκης

Ψηφιακές Επικοινωνίες

CRC

REPORT

Όνομα: Χρήστος-Αλέξανδρος Τσιγγιρόπουλος
ΑΕΜ: 3872

Εκφώνηση Άσκησης (.pdf)

Γλώσσα Προγραμματισμού C

1. Εισαγωγή

Στόχος της εργασίας είναι η ανάπτυξη κώδικα, που υλοποιεί τον αλγόριθμο ανίχνευσης σφαλμάτων CRC. CRC (Cyclic Redundancy Check) ή αλλιώς Κυκλικοί κώδικες, χρησιμοποιούνται στην ανίχνευση σφαλμάτων μεταδιδόμενης πληροφορίας.

Η Εργασία έχει υλοποιηθεί χρησιμοποιώντας την γλώσσα προγραμματισμού C.

Ο αλγόριθμος που αναπτύχθηκε είναι simulation αυτής της μετάδοσης, με ορισμένο bit error rate (BER) το 0.001. Επειδή δεν έχουμε εξωτερικά σφάλματα και η λογική αυτού του simulation είναι να δούμε πόσο καλά λειτουργεί ο αλγόριθμος CRC έχει οριστεί ένα σχετικά μεγάλο BER*.

Το αρχείο περιέχει ένα αρχείο κώδικα C και το εκτελέσιμό του:

- CRC.c
- CRC.exe

Τέλος, για να είναι αξιόπιστα τα αποτελέσματα, δημιουργήθηκε δείγμα ενός δισεκατομμυρίου τυχαίων μηνυμάτων των 20 bits έκαστο.

BER* = Είναι η πιθανότητα 1->0 και το 0->1 (δλδ η πιθανότητα σφάλματος σε ένα bit).

2. Παράδειγμα Λειτουργίας

Το πρόγραμμα ξεκινάει χωρίς να δέχεται καμία είσοδο. Αυτό συμβαίνει αφού γνωρίζουμε εκ των προτέρων τον αριθμό του καθαρού μηνύματος k = 20 bits, την προκαθορισμένη σταθερα P = 110101 και το BER = 0.001 . Το μέγεθος n του τελικού μεταδιδόμενου μηνύματος, είναι n = k + p - 1 , με p = το μέγεθος της σταθεράς P. Δηλαδή το n = 20 + 6 - 1 = 25 bits.

Οπότε κατευθείαν το πρόγραμμα αρχίζει να κάνει προσομοίωση για 1 δισεκατομμύριο μηνύματα των 20 bits (Μέσο ενός For loop).

Για κάθε μήνυμα ξεχωριστά :

1. Δημιουργούμε την προς μετάδοση ακολουθία δεδομένων D των 20 bits, που το κάθε bit αποτελείται ισοπίθανα απο 0 ή 1.

2. Δημιουργούμε την ακολουθία D2, βάζοντας n-k μηδενικά στο τέλος της ακολουθίας D .

3. Δημιουργούμε την ακολουθία F των n-k bits, που είναι το υπόλοιπο της xor διαίρεσης του D2 / P .

4. Δημιουργούμε την τελική ακολουθία T των n bits, βάζοντας την F στο τέλος της D2 . Δηλαδή αντικαθιστούμε τις τελευταίες n-k μηδενικές θέσεις της D2 ακολουθίας με τα στοιχεία της ακολουθίας F .

5. Δημιουργούμε την ακολουθία T’ του «αλλοιωμένου» μηνύματος μετά την μετάδοση μέσω ενός ενόρυβου καναλιού . Η T΄ προέρχεται από την T , με κάθε bit της T’ , να έχει πιθανότητα 0.001 (BER) να είναι απαλλαγμένο από αυτό της T . Αν έχει γίνει έστω και μία αλλαγή, αυξάνουμε τον μετρητή των αλλοιωμένων μηνυμάτων κατα 1.

6. Τέλος, ελέγχουμε την ακολουθία T’ με τον αλγόριθμο CRC . Αν διαπιστωθεί ότι υπάρχει σφάλμα, τότε αυξάνουμε τον μετρητή των μηνυμάτων που έχουν βρεθεί με σφάλμα .

Το αποτέλεσμα του αλγορίθμου είναι, το ποσοστό των μηνυμάτων που φτάνουν με σφάλμα, το ποσοστό των μηνυμάτων που ανιχνεύονται ως εσφαλμένα απο

το CRC και το ποσοστό που φτάνουν με σφάλμα και δεν αναγνωρίζονται απο το CRC.

3. Υλοποίηση

Σε αυτή την ενότητα θα γίνει αναφορά για τα σημαντικότερα σημεία του κώδικα. Τα περισσότερα απο αυτά, αφορούν τις συναρτήσεις που χρησιμοποιούνται για την δημιουργία των ακολουθιών D, D2, F, T, T' αλλά και για τον έλεγχο CRC.

1. Για την δημιουργία της ακολουθίας D χρησιμοποιούμε την συνάρτηση .

```
/** Input : *String = address of a string with size = k
 *  Initializes with the same posibility every char of the string with '0' or '1'
 *  Return : void
 */
void create_Message(char* D){
    for(i = 0; i < k; i++ ){
        double foo = (double)rand()/(double)RAND_MAX;
        D[i] = foo>0.5 ? '1' : '0';
    }
}
```

Αυτή, δέχεται ως είσοδο μια διεύθυνση ενός πίνακα χαρακτήρων (String) που έχει μέγεθος k και αρχικοποιεί το string αυτό με ισοπίθανα 0,1 σε κάθε θέση.

2. Όσον αφορά την δημιουργία της D2 γίνεται χρήση της συνάρτησης `void get_2nkD(char *,const char*, int);` Οι είσοδοι είναι μια διεύθυνση που αποθηκεύει το D2 μεγέθους n, μια διεύθυνση που δείχνει στην ακολουθία D (του προηγούμενου ερωτήματος) και έναν ακέραιο n που δείχνει το μέγεθος του D2. Αυτό που κάνει η συνάρτηση αυτή είναι, να αποθηκεύει στις πρώτες k θέσεις του D2 το D και στις τελευταίες n-k το 0. Αυτό υλοποιείται με τον παρακάτω κώδικα.

```
void get_2nkD(char *D_2nk, const char* D , int n) {
    for (i = 0; i < n; i++) {
        if (i < k)
            D_2nk[i] = D[i];
        else
            D_2nk[i] = '0';
    }
}
```

3. Ο κώδικας της συνάρτησης που χρησιμοποιείται για την δημιουργία της ακολουθίας F είναι ο ακόλουθος. Στη συνάρτηση αυτή χρησιμοποιείται και μία βοηθητική συνάρτηση `int subtract_mod2(char* ,const char* ,int);`. Η συνάρτηση αυτή υλοποιεί μια απλή xor διαίρεση ,το πρώτο string είναι ο διαιρετέος , το δεύτερο ο διαιρέτης και το int είναι το μέγεθος των strings. Αφού υλοποιηθεί η διαιρεση, στον διαιρετέος έχει αποθηκευτεί το υπόλοιπο και η συνάρτηση επιστρέφει πόσα bits «έχασε» από τη διαίρεση.

   Ο παρακάτω κώδικα δημιουργεί μια νέα ακολουθία temp που για στοιχεία της έχει το υπόλοιπο της XOR διαίρεσης μεταξύ των ακολουθιών D2 / P . Αυτό γίνεται στην διπλή while loop. Το temp χρησιμοποιείται ως διαιρετέος, το p ώς διαιρέτης και μετά από το κάλεσμα της παραπάνω βοηθητικής συνάρτησης το temp έχει το υπόλοιπο. Ανανεώνουμε την συμβολοσειρά temp (2η loop) και ξανα καλούμε την βοηθητική συνάρτηση. Η διαδικασία ολοκληρώνεται όταν δεν έχουμε άλλα στοιχεία για να ανανεώσουμε το temp , όταν δηλαδή ολοκληρωθεί η διαίρεση xor . Τότε το υπόλοιπο βρίσκεται στη μεταβλητή temp όπου είναι και η επιστρεφόμενη συμβολοσειρά.

```
char* get_R(const char* D_2nk, int n,char* P,int size_p,int* size_of_r){
    int count = 0 , c1 = 0;
    char *temp = malloc( Size: size_p* sizeof(char));
    for(i=0;i<size_p;i++)
        temp[i]=D_2nk[i];

    while(n-count>=size_p){
        c1 = subtract_mod2(temp, p: P, size_of_p: size_p);
        count += c1;

        i = - c1 ;
        while(1){
            if(i == 0 || count+size_p+i == n)
                break;
            temp[size_p+i] = D_2nk[count+size_p+i];
            i++;
        }
    }
    count = 0;
    for(i=size_p-1;i>=0;i--){
        if(temp[i]=='-')
            count ++;
        else
            break;
    }

    *size_of_r = size_p-count;
    return temp;
}
```

4. `char* get_T(const char* D2,int D2_s, const char* F, int F_s);` είναι η συνάρτηση που χρησιμοποιείται για την δημιουργία της T ακολουθίας. Έχει 4 ορίσματα τις ακολουθίες D2 και F και τα μεγέθη τους. Αυτό που κάνει είναι να δημιουργεί μια νέα συμβολοσειρά και να αρχικοποιεί τις πρώτες k θέσεις της με το D2 και στις τελευταίες n-k με το F.

```
char* get_T(const char* D_2nk,int size_of_D2nk, const char* FSC, int size_of_FCS){
    char * T = malloc( Size: size_of_D2nk * sizeof(char));
    for(i = 0 ; i < size_of_D2nk ; i++ )
        T[i] = D_2nk[i];

    for(i = 1; i <= size_of_FCS; i++)
        T[size_of_D2nk - i] = FSC[size_of_FCS -i];

    return T ;
}
```

5. Η ακολουθία T΄ του αλλοιωμένου μηνύματος δημιουργείται με την βοήθεια της συνάρτησης : `int BitErrorRate(char* T,int s_T ,double ber);`
   Είσοδος : T μήνυμα προς μετάδοση, μέγεθος του T και το BER (Bit Error Rate). Χρησιμοποιείται η random συνάρτηση για να λάβουμε μια τιμή μεταξύ [0-1]. Για κάθε char του T αν random < ber τότε αλλάζουμε το char. Η αλλαγμένη T που προκύπτει είναι η T΄ που είναι το λαμβανόμενο μήνυμα του δέκτη. Η συνάρτηση επιστρέφει 1 αν έγινε τουλάχιστον μια αλλαγή διαφορετικά 0.

```
int BitErrorRate(char* T,int size_of_T ,double ber){
    int flag = 0;

    for(i=0;i<size_of_T;i++){
        double foo = (double)rand()/(double)RAND_MAX;   // foo = [0..1]
        if( foo < ber ){
            T[i] = T[i]=='0' ? '1' : '0';
            flag++ ;
        }
    }
    return flag>0 ? 1 : 0;
}
```

6. Τέλος, η συνάρτηση `int CRS(char* T, int sT, char* P, int sp)` είναι αυτή που ελέγχει αν το μήνυμα έχει λάθος. Αυτο το επιτυγχάνει κάνοντας την XOR διαίρεση T / P και αν το υπόλοιπο αυτής είναι 0 τότε δεν βρέθηκε κάποιο λάθος, διαφορετικά το μήνυμα είναι σίγουρα αλλοιωμένο. Το υπόλοιπο της XOR διαίρεσης, υλοποιείται από την συνάρτηση get_R(...)

. Το αποτέλεσμα αυτής, το αποθηκεύουμε σε μια νέα μεταβλητή. Αν αυτή είναι 0, τότε επιστρέφουμε 1 (δεν βρέθηκε λαθος) και 0 διαφορετικά.

```
int CRS(char* T, int size_of_T, char* P, int size_of_p){
    int int_temp;
    char *temp = get_R( D_2nk: T,  n: size_of_T, P, size_p: size_of_p, size_of_r: &int_temp);

    for (i=0;i<int_temp;i++){
        if(temp[i] != '0' && temp[i] !='-'){
            free( Memory: temp);
            return 0;                          // Something is Wrong
        }
    }
    free( Memory: temp);
    return 1;                                  // Everything is Good
}
```

4. Αποτέλεσμα Και Σχολιασμός

```
Done = 79.7%
Done = 79.8%
Done = 79.9%
Done = 80.0%
Done = 80.1%
Done = 80.2%
Done = 80.3%
Done = 80.4%
```

Κατά την διάρκεια εκτέλεσης του προγράμματος εμφανίζεται η κατάσταση της προσομοίωσης όπως φαίνεται στην διπλανή φωτογραφία.

Για να έχουμε αξιόπιστα δεδομένα έγινε εκτέλεση του προγράμματος 3 φορές.

Τα αποτελέσματα για 1 δις. μηνύματα με P = 110101, BER = 0.001 και k = 20 , είναι τα εξής :

```
The results are:


Messages that got an error: 2.4874914%
Messages found to have an error from CRC: 2.4865007%

Percentage of messages that have an error and are not traced from CRC : 0.03982727%
```

```
The results are:


Messages that got an error: 2.4876108%
Messages found to have an error from CRC: 2.4866227%

Percentage of messages that have an error and are not traced from CRC : 0.03972084%
```

```
The results are:


Messages that got an error: 2.4874919%
Messages found to have an error from CRC: 2.4865021%

Percentage of messages that have an error and are not traced from CRC : 0.03979108%
```

Σύμφωνα με τα παραπάνω αποτελέσματα, βγάζουμε τον παρακάτω πίνακα με τους μέσους όρους των ποσοστών.

Κατηγορία Μηνυμάτων	Ποσοστό μηνυμάτων (%)
Εσφαλμένα μηνύματα	2.48753 %
Εσφαλμένα μηνύματα που εντόπισε ο CRC	2.48654 %
Εσφαλμένα μηνύματα που δεν εντοπίστηκαν	0.03978 %

Βλέπουμε ότι τα εσφαλμένα μηνύματα είναι περίπου 2.5% που είναι ένα αναμενόμενο ποσοστό. Αυτό γιατί, η πιθανότητα ένα μήνυμα να έχει τουλάχιστον ένα λάθος, είναι ίση με τον αριθμό των bits του επί το ber. Στο παράδειγμα μας, αυτή η πιθανότητα είναι 25 bits* 0.001 = 0.025 = 2.5%. Οπότε, για 1 δις. μηνύματα , τα εσφαλμένα θα είναι περίπου 2.5%.

Επιπλέον, όπως φαίνεται από τον πίνακα, ο αλγόριθμος CRC δεν μπόρεσε να ανιχνεύσει μόλις το 0.04 % των εσφαλμένων μηνυμάτων. Συνεπώς, γίνεται εύκολα αντιληπτό πως αυτός ο αλγόριθμος είναι εξαιρετικά αξιόπιστος και αποδοτικός στο να εντοπίζει σφάλματα, που δημιουργήθηκαν π.χ. κατά την μετάδοση πληροφορίας μέσα από ένα κανάλι με θόρυβο.

Το συνολικό project μαζί με αυτή την αναφορά μπορεί να βρεθεί στο github.