Department of Computer Science and Engineering

S.G.Shivanirudh , 185001146, Semester V

16 September 2020

UCS1511 - Networks Laboratory

Exercise 8: Hamming Code

Objective:

Construct **Hamming Code** for a given binary data.

Code:

Hamming functions:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
```

```
#include<unistd.h>
#include<arpa/inet.h>

int power(int num, int exp){
    int pdt = 1;
    while(exp--){
        pdt *= num;
    }
    return pdt;
}

void strrev(char* s){
    int len = strlen(s);
    int i = 0;
    int j = len-1;
    while(i<j){
        char tmp = s[i];
        s[i] = s[j];
        s[j] = tmp;
        i++;
        j--;
    }
}

int checkBinary(char *code){
    int check = 1;
    for(int i = 0; code[i] && check; i++){
        if(code[i] != ’1’ && code[i] != ’0’)
            check = 0;
    }
    return check;
}

char* conv_to_bin(int number){
    char *bin = (char*)calloc(100, sizeof(char));
    int n = number;
    int pos=0;
    while(n>0){
        bin[pos++] = (’0’+(n%2));
        n /= 2;
    }
    bin[pos] = ’\0’;
    strrev(bin);
    return bin;
}
```

```

int conv_to_dec(char *number){
    int num = 0;
    char *copy = (char*)calloc(100, sizeof(char));
    strcpy(copy, number);
    for(int i = 0; copy[i]; i++){
        if(copy[i] == ’1’)
            num += power(2, i);
    }
    printf("\n%s %d", number, num);
    return num;
}

int check_position(int number, int position){
    char *bin=(char*)calloc(100, sizeof(char));
    strcpy(bin, conv_to_bin(number));
    int len = strlen(bin);
    return (bin[len - position]==’1’)? 1 : 0;
}

char* even_parity(char *input){
    char *ecode = (char*)calloc(100, sizeof(char));
    int ip_len = strlen(input);
    int red_bits = 0;
    for(int i = 0; i<100; i++){
        int lhs = power(2.0, i);
        int rhs = ip_len + i + 1;
        if( lhs >= rhs){
            red_bits = i;
            break;
        }
    }
    char *ip = (char*)calloc(100, sizeof(char));
    strcpy(ip, input);
    strrev(ip);
    int code_len = ip_len + red_bits;
    //Assign data bits
    int ip_ctr = 0;
    for(int i = 0;i<code_len;i++){
        int ham_bit = 0;
        for(int j = 0; j < code_len && !ham_bit; j++){
            if((i+1) == power(2, j))
                ham_bit = 1;
        }
        if(ham_bit){
```

```c
            ecode[i] = '0';
        }
        else{
            ecode[i] = ip[ip_ctr];;
            ip_ctr++;
        }
    }

    //Hamming code
    int pos = 0; //Position to check in binary value
    for(int i = 0;i<code_len;i++){
        int ham_bit = 0;
        for(int j = 0; j < code_len && !ham_bit; j++){
            if((i+1) == power(2, j)){
                ham_bit = 1;
                pos += 1;
            }
        }
        if(ham_bit){
            int ctr = 0;
            for(int j = 0;j<code_len;j++){
                int check_pos = check_position(j+1, pos);
                if(ecode[j] == '1'&&check_pos){
                    ctr++;
                }
            }
            ecode[i] = ctr%2? '1':'0';
        }
    }
    //Reversing code
    strrev(ecode);
    return ecode;
}

char *odd_parity(char *input){
    char *ocode = (char*)calloc(100, sizeof(char));

    int ip_len = strlen(input);
    int red_bits = 0;
    for(int i = 0; i<100; i++){
        int lhs = power(2.0, i);
        int rhs = ip_len + i + 1;
        if( lhs >= rhs){
            red_bits = i;
            break;
        }
    }
    char *ip = (char*)calloc(100, sizeof(char));
    strcpy(ip, input);
    strrev(ip);
    int code_len = ip_len + red_bits;
    //Assign data bits
    int ip_ctr = 0;
    for(int i = 0;i<code_len;i++){
        int ham_bit = 0;
        for(int j = 0; j < code_len && !ham_bit; j++){
            if((i+1) == power(2, j))
                ham_bit = 1;
        }
        if(ham_bit){
            ocode[i] = '0';
        }
        else{
            ocode[i] = ip[ip_ctr];
            ip_ctr++;
        }
    }

    //Hamming code
    int pos = 0; //Position to check in binary value
    for(int i = 0;i<code_len;i++){
        int ham_bit = 0;
        for(int j = 0; j < code_len && !ham_bit; j++){
            if((i+1) == power(2, j)){
                ham_bit = 1;
                pos += 1;
            }
        }
        if(ham_bit){
            int ctr = 0;
            for(int j = 0;j<code_len;j++){
                int check_pos = check_position(j+1, pos);
                if(ocode[j] == '1'&&check_pos){
                    ctr++;
                }
            }
            ocode[i] = ctr%2? '0':'1';
        }
    }

    //Reversing code
    strrev(ocode);
    return ocode;
}

int compute_error_pos(char *code, int parity){
    int code_len = strlen(code);
    char *value = (char*)calloc(100, sizeof(char));

    int red_bits = 0, ip_len = 0;
    for(int i = 0;i<code_len; i++){
        ip_len = code_len - i;
        int lhs = power(2, i);
        int rhs = ip_len + i + 1;
        if(lhs >= rhs){
            red_bits = i;
            break;
        }
    }
    for(int i =0; i<red_bits; i++){
        value[i] = ’0’;
    }
    int vctr = 0;
    int pos = 0;
    if(parity){
        for(int i = 0;i<code_len;i++){
            int ham_bit = 0;
            for(int j = 0; j < code_len && !ham_bit; j++){
                if((i+1) == power(2, j)){
                    ham_bit = 1;
                    pos += 1;
                }
            }
            if(ham_bit){
                int ctr = 0;
                for(int j = 0;j<code_len;j++){
                    int check_pos = check_position(j+1, pos);
                    if(code[j] == ’1’&&check_pos){
                        ctr++;
                    }
                }
                value[vctr++] = ctr%2? ’0’:’1’;
            }
        }
    }
```

```c
    else{
        for(int i = 0;i<code_len;i++){
            int ham_bit = 0;
            for(int j = 0; j < code_len && !ham_bit; j++){
                if((i+1) == power(2, j)){
                    ham_bit = 1;
                    pos += 1;
                }
            }
            if(ham_bit){
                int ctr = 0;
                for(int j = 0;j<code_len;j++){
                    int check_pos = check_position(j+1, pos);
                    if(code[j] == ’1’&&check_pos){
                        ctr++;
                    }
                }
                value[vctr++] = ctr%2? ’1’:’0’;
            }
        }
    }
    strrev(value);
    return conv_to_dec(value);
}

char* decode(char *code){
    char *data = (char*)calloc(100, sizeof(char));

    int code_len = strlen(data);
    int red_bits = 0, ip_len = 0;
    for(int i = 0; i<100; i++){
        ip_len = code_len - i;
        int lhs = power(2.0, i);
        int rhs = ip_len + i + 1;
        if( lhs >= rhs){
            red_bits = i;
            break;
        }
    }

    char *copy = (char*)calloc(100, sizeof(char));
    strcpy(copy, code);
    //Hamming code
    int pos = 0; //Position to check in binary value
    for(int i = 0;i<code_len;i++){
        int ham_bit = 0;
        for(int j = 0; j < code_len && !ham_bit; j++){
            if((i+1) == power(2, j)){
                ham_bit = 1;
                pos += 1;
            }
        }
        if(ham_bit){
            continue;
        }
        else{
            strcat(data, copy[i]);
        }
    }

    //Reversing code
    strrev(data);
    return data;
}
```

Server:

```c
#include "Hamming.h"

int main(int argc, char **argv){
    if(argc > 1){
        perror("Error:No need arguments for server");
        exit(1);
    }
    struct sockaddr_in server, client;
    pid_t child;
    char buffer[1024];

    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if(sockfd < 0){
        perror("\nError: Socket");
        exit(1);
    }

    bzero(&server, sizeof(server));

    server.sin_family = AF_INET;
    server.sin_port = htons(7000);
    server.sin_addr.s_addr = INADDR_ANY;

    if(bind(sockfd, (struct sockaddr*)&server, sizeof(server)) < 0){
        perror("Error: Bind");
        exit(1);
    }

    listen(sockfd, 5);
    int len = sizeof(client);

    while(1){
        int newfd = accept(sockfd, (struct sockaddr*)&client, &len);
        if(newfd < 0){
            perror("Error:Accept");
            exit(1);
        }
        socklen_t port;
        struct sockaddr_in addr;
        int res = getpeername(newfd, (struct sockaddr*)&addr, &port);
        printf("\nClient %d", ntohs(port));

        child = fork();
        if(child == 0){
            while(1){
                read(newfd, buffer, sizeof(buffer));
                printf("\nTransmitted data: %s\n", buffer);

                int error_pos = compute_error_pos(buffer, 0);

                if(error_pos>0){
                    printf("\nError at %d bit", error_pos);
                    strrev(buffer);
                    char c = buffer[error_pos - 1];
                    buffer[error_pos-1] = c == '1'? '0':'1';
                    strrev(buffer);
                    printf("\nAfter correction: %s", buffer);
                }
                strcpy(buffer, decode(buffer));

                printf("\nOriginal data: %s\n", buffer);
                write(newfd, buffer, sizeof(buffer));
            }
            close(newfd);
        }
    }

}
```

Hamming functions:

```c
#include "Hamming.h"

int main(int argc, char **argv){
    if(argc != 2){
        perror("Error:Server ip needed for client");
        exit(1);
    }
    struct sockaddr_in server;
    char buffer[1024];

    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if(sockfd < 0){
        perror("\nError: Socket");
        exit(1);
    }

    bzero(&server, sizeof(server));

    server.sin_family = AF_INET;
    server.sin_port = htons(7000);
    server.sin_addr.s_addr = inet_addr(argv[1]);

    connect(sockfd, (struct sockaddr*)&server, sizeof(server));

    while(1){
        int pos = 0;
        printf("\nEnter input data: ");scanf(" %[^\n]", buffer);
        printf("Enter position for error: ");scanf("%d", &pos);
        strcpy(buffer, even_parity(buffer));
        if(pos >=0 ){
            strrev(buffer);
            char c = buffer[pos];
            buffer[pos - 1] = c == ’1’? ’0’:’1’;
            strrev(buffer);
        }
        printf("\nTransmitted data:%s", buffer);
        write(sockfd, buffer, sizeof(buffer));

        bzero(&buffer, sizeof(buffer));
        read(sockfd, buffer, sizeof(buffer));
        printf("\nCorrect data: %s", buffer);
    }
}
```

Output:

Server:

```
Client 4096
Transmitted data: 10101101110

Error at 6 bit
After correction: 10101001110
Original data: 1011001
```

Client:

```
Enter input data: 1011001
Enter position for error: 6

Transmitted data: 10101101110
Correct data: 1011001
```