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>
```

```c
#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;
}
```

```
52
53 int conv_to_dec(char *number){
54     int num = 0;
55     char *copy = (char*)calloc(100, sizeof(char));
56     strcpy(copy, number);
57     for(int i = 0; copy[i]; i++){
58         if(copy[i] == '1')
59             num += power(2, i);
60     }
61     printf("\n%s %d", number, num);
62     return num;
63 }
64
65 int check_position(int number, int position){
66     char *bin=(char*)calloc(100, sizeof(char));
67     strcpy(bin, conv_to_bin(number));
68     int len = strlen(bin);
69     return (bin[len - position]=='1')? 1 : 0;
70 }
71
72 char* even_parity(char *input){
73     char *ecode = (char*)calloc(100, sizeof(char));
74     int ip_len = strlen(input);
75     int red_bits = 0;
76     for(int i = 0; i<100; i++){
77         int lhs = power(2.0, i);
78         int rhs = ip_len + i + 1;
79         if( lhs >= rhs){
80             red_bits = i;
81             break;
82         }
83     }
84     char *ip = (char*)calloc(100, sizeof(char));
85     strcpy(ip, input);
86     strrev(ip);
87     int code_len = ip_len + red_bits;
88     //Assign data bits
89     int ip_ctr = 0;
90     for(int i = 0;i<code_len;i++){
91         int ham_bit = 0;
92         for(int j = 0; j < code_len && !ham_bit; j++){
93             if((i+1) == power(2, j))
94                 ham_bit = 1;
95         }
96         if(ham_bit){
```

```
97              ecode[i] = '0';
98          }
99          else{
100             ecode[i] = ip[ip_ctr];;
101             ip_ctr++;
102         }
103     }
104
105     //Hamming code
106     int pos = 0; //Position to check in binary value
107     for(int i = 0;i<code_len;i++){
108         int ham_bit = 0;
109         for(int j = 0; j < code_len && !ham_bit; j++){
110             if((i+1) == power(2, j)){
111                 ham_bit = 1;
112                 pos += 1;
113             }
114         }
115         if(ham_bit){
116             int ctr = 0;
117             for(int j = 0;j<code_len;j++){
118                 int check_pos = check_position(j+1, pos);
119                 if(ecode[j] == '1'&&check_pos){
120                     ctr++;
121                 }
122             }
123             ecode[i] = ctr%2? '1':'0';
124         }
125     }
126     //Reversing code
127     strrev(ecode);
128     return ecode;
129 }
130
131 char *odd_parity(char *input){
132     char *ocode = (char*)calloc(100, sizeof(char));
133
134     int ip_len = strlen(input);
135     int red_bits = 0;
136     for(int i = 0; i<100; i++){
137         int lhs = power(2.0, i);
138         int rhs = ip_len + i + 1;
139         if( lhs >= rhs){
140             red_bits = i;
141             break;
```

```
142         }
143     }
144     char *ip = (char*)calloc(100, sizeof(char));
145     strcpy(ip, input);
146     strrev(ip);
147     int code_len = ip_len + red_bits;
148     //Assign data bits
149     int ip_ctr = 0;
150     for(int i = 0;i<code_len;i++){
151         int ham_bit = 0;
152         for(int j = 0; j < code_len && !ham_bit; j++){
153             if((i+1) == power(2, j))
154                 ham_bit = 1;
155         }
156         if(ham_bit){
157             ocode[i] = '0';
158         }
159         else{
160             ocode[i] = ip[ip_ctr];
161             ip_ctr++;
162         }
163     }
164 
165     //Hamming code
166     int pos = 0; //Position to check in binary value
167     for(int i = 0;i<code_len;i++){
168         int ham_bit = 0;
169         for(int j = 0; j < code_len && !ham_bit; j++){
170             if((i+1) == power(2, j)){
171                 ham_bit = 1;
172                 pos += 1;
173             }
174         }
175         if(ham_bit){
176             int ctr = 0;
177             for(int j = 0;j<code_len;j++){
178                 int check_pos = check_position(j+1, pos);
179                 if(ocode[j] == '1'&&check_pos){
180                     ctr++;
181                 }
182             }
183             ocode[i] = ctr%2? '0':'1';
184         }
185     }
186 
```

```
187     //Reversing code
188     strrev(ocode);
189     return ocode;
190 }
191
192 int compute_error_pos(char *code, int parity){
193     int code_len = strlen(code);
194     char *value = (char*)calloc(100, sizeof(char));
195
196     int red_bits = 0, ip_len = 0;
197     for(int i = 0;i<code_len; i++){
198         ip_len = code_len - i;
199         int lhs = power(2, i);
200         int rhs = ip_len + i + 1;
201         if(lhs >= rhs){
202             red_bits = i;
203             break;
204         }
205     }
206     for(int i =0; i<red_bits; i++){
207         value[i] = '0';
208     }
209     int vctr = 0;
210     int pos = 0;
211     if(parity){
212         for(int i = 0;i<code_len;i++){
213             int ham_bit = 0;
214             for(int j = 0; j < code_len && !ham_bit; j++){
215                 if((i+1) == power(2, j)){
216                     ham_bit = 1;
217                     pos += 1;
218                 }
219             }
220             if(ham_bit){
221                 int ctr = 0;
222                 for(int j = 0;j<code_len;j++){
223                     int check_pos = check_position(j+1, pos);
224                     if(code[j] == '1'&&check_pos){
225                         ctr++;
226                     }
227                 }
228                 value[vctr++] = ctr%2? '0':'1';
229             }
230         }
231     }
```

```
    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++){
```

```
277         int ham_bit = 0;
278         for(int j = 0; j < code_len && !ham_bit; j++){
279             if((i+1) == power(2, j)){
280                 ham_bit = 1;
281                 pos += 1;
282             }
283         }
284         if(ham_bit){
285             continue;
286         }
287         else{
288             strcat(data, copy[i]);
289         }
290     }
291 
292     //Reversing code
293     strrev(data);
294     return data;
295 }
```

Server:

```
1  #include "Hamming.h"
2  
3  int main(int argc, char **argv){
4      if(argc > 1){
5          perror("Error:No need arguments for server");
6          exit(1);
7      }
8      struct sockaddr_in server, client;
9      pid_t child;
10     char buffer[1024];
11 
12     int sockfd = socket(AF_INET, SOCK_STREAM, 0);
13     if(sockfd < 0){
14         perror("\nError: Socket");
15         exit(1);
16     }
17 
18     bzero(&server, sizeof(server));
19 
20     server.sin_family = AF_INET;
```

```
21      server.sin_port = htons(7000);
22      server.sin_addr.s_addr = INADDR_ANY;
23
24      if(bind(sockfd, (struct sockaddr*)&server, sizeof(server)
        ) < 0){
25          perror("Error: Bind");
26          exit(1);
27      }
28
29      listen(sockfd, 5);
30      int len = sizeof(client);
31
32      while(1){
33          int newfd = accept(sockfd, (struct sockaddr*)&client,
        &len);
34          if(newfd < 0){
35              perror("Error:Accept");
36              exit(1);
37          }
38          socklen_t port;
39          struct sockaddr_in addr;
40          int res = getpeername(newfd, (struct sockaddr*)&addr,
        &port);
41          printf("\nClient %d", ntohs(port));
42
43          child = fork();
44          if(child == 0){
45              while(1){
46                  read(newfd, buffer, sizeof(buffer));
47                  printf("\nTransmitted data: %s\n", buffer);
48
49                  int error_pos = compute_error_pos(buffer, 0);
50
51                  if(error_pos>0){
52                      printf("\nError at %d bit", error_pos);
53                      strrev(buffer);
54                      char c = buffer[error_pos - 1];
55                      buffer[error_pos-1] = c == '1'? '0':'1';
56                      strrev(buffer);
57                      printf("\nAfter correction: %s", buffer);
58                  }
59                  strcpy(buffer, decode(buffer));
60
61                  printf("\nOriginal data: %s\n", buffer);
62                  write(newfd, buffer, sizeof(buffer));
```

```
63              }
64              close(newfd);
65          }
66      }
67
68  }
```

Hamming functions:

```
1   #include "Hamming.h"
2
3   int main(int argc, char **argv){
4       if(argc != 2){
5           perror("Error:Server ip needed for client");
6           exit(1);
7       }
8       struct sockaddr_in server;
9       char buffer[1024];
10
11      int sockfd = socket(AF_INET, SOCK_STREAM, 0);
12      if(sockfd < 0){
13          perror("\nError: Socket");
14          exit(1);
15      }
16
17      bzero(&server, sizeof(server));
18
19      server.sin_family = AF_INET;
20      server.sin_port = htons(7000);
21      server.sin_addr.s_addr = inet_addr(argv[1]);
22
23      connect(sockfd, (struct sockaddr*)&server, sizeof(server)
        );
24
25      while(1){
26          int pos = 0;
27          printf("\nEnter input data: ");scanf(" %[^\n]",
        buffer);
28          printf("Enter position for error: ");scanf("%d", &pos
        );
29          strcpy(buffer, even_parity(buffer));
30          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
```
