

확률 및 통계

Assignment #5: The Context Adaptive Huffman Coding

학 과: 컴퓨터정보공학부

담당교수님: 심동규 교수님

학 번: 2015722068

이 름: 남윤창

1. Introduction

본 과제는 Assignment #4에서 구현한 Huffman Coding을 확장하여 Context Adaptive Huffman Coding을 구현한다. Huffman Encoder, Decoder를 각각 작성한다. Encoder는 training_input.txt를 Assignment #4와 마찬가지로 Huffman Table을 생성한다. 추가로 조건부 확률을 이용하여 Context Adaptive Huffman Table을 작성한다. 이를 이용하여 Encoding을 진행하고 Cost를 측정한다. Minimal Cost를 찾게 되면 각각 huffman_table.hbs, context_adaptive_huffman_table.hbs로 인코딩 하여 저장한다. 그리고 나머지 test input들도 인코딩을 진행하여 각각 저장한다. Decoder는 앞서 생성한 테이블을 읽어 역으로 허프만 테이블을 생성하고, 인코딩 된 파일들을 디코딩 하여 원본으로 복구하는 작업을 진행한다.

Assignment #4와 다르게, training_input으로 허프만 테이블을 작성하기 때문에 빈도수가 측정이 되지 않은 문자들에 대해서도 테이블을 작성한다.

Assignment #4와 마찬가지로 Stuffing byte, EOD, byte align을 고려하여 작성한다.

모든 코드가 끝나게 되면 $Cost = C + 0.001 * table_bytes$ 를 통해 압축이 얼마나 효율적으로 진행되었는지 비용을 계산하여 분석한다.

2. Source Code

기본적인 내용은 Assignment #4와 동일하므로 Assignment #5에 추가된 내용을 위주로 설명된다.

Encoder.cpp

```
// Context Adaptive Node
typedef struct AdaptiveHuffmanTreeNode {
    string encoded;
    int freq;
    unsigned char preceding_symbol; // preceding_symbol : X-1
    unsigned char ascii_code;      // current_symbol   : X
    AdaptiveHuffmanTreeNode* leftnode;
    AdaptiveHuffmanTreeNode* rightnode;
} AdaptiveHuffmanTreeNode;
```

조건부 허프만 테이블을 작성하기 위해 새로운 구조체를 생성하여 preceding symbol의 값과 현재 문자를 저장하여 X-1 -> X의 관계를 나타내도록 생성했다.

```

while (1) {
    chTemp = fgetc(input); //read current character
    if (feof(input)) { //file end -> break
        ascii_adaptive[chPrev][end_of_data]++;
        break;
    }
    ascii[chTemp]++; //count up - normal huffman
    if (chPrev != NULL)
        ascii_adaptive[chPrev][chTemp]++; //count up - adaptive huffman
    chPrev = chTemp;
}
// Step 1.2 End of Data 설정
ascii[end_of_data]++;

```

Training_input을 읽어 ascii_adaptive라는 2차원 배열을 생성하여 preceding symbol과 현재 데이터의 개수를 세어주었다.

Assignment #4에 있는 코드를 그대로 가져와서 사용하였다. Normal Huffman Table을 먼저 작성했다.

```

// ===== Step 3. Create Context Adaptive Huffman Table ===== //
int atnum = 0;
for (int i = 0; i < 128; i++) {
    // Step 3.1 Create Min-Heap of each preceding symbol (X-1)
    priority_queue< pair<int, AdaptiveHuffmanTreeNode*>, vector< pair<int, AdaptiveHuffmanTreeNode*>>, greater< pair<int, AdaptiveHuffmanTreeNode*>>> > apq;
    for (int j = 0; j < 128; j++) {
        if (ascii[i] > 300) { // parameter for minimal cost
            AdaptiveHuffmanTreeNode * at;
            at = new AdaptiveHuffmanTreeNode;
            at->preceding_symbol = i; //이전 문자
            at->ascii_code = j; //현재 문자
            at->freq = ascii_adaptive[i][j]; //빈도수
            at->leftnode = NULL;
            at->rightnode = NULL;
            apq.push(make_pair(ascii_adaptive[i][j], at)); //min-heap
        }
    }
}

```

Training_input을 읽어 빈도수에 대해 특정 빈도수 이상의 데이터에 대해서만 조건부 허프만 테이블을 생성하도록 min-heap에 추가한다.

```

}

// Step 3.3 Add encoded number
string a_encoded_num;
A_Inorder_traversal(a_root, a_encoded_num, atnum);

```

```

// Adaptive Tree -> In-order Traversal
void A_Inorder_traversal(AdaptiveHuffmanTreeNode* t, string num, int &tnum) {
    if (t != NULL) {
        A_Inorder_traversal(t->leftnode, num + "0", tnum); //왼쪽 + 0
        if (t->ascii_code != NULL) {
            if (num == "") num = "0"; //만약 N/A라면 0으로 인코딩
            t->encoded = num;
            tnum = MakeTable_Adaptive(t, tnum);
        }
        A_Inorder_traversal(t->rightnode, num + "1", tnum); //오른쪽 + 0
    }
}

```

중위순회 하며 조건부 허프만 테이블을 생성한다. MakeTable_Adaptive함수는 기존 함수에 Preceding symbol만을 추가하였다.

preceding symbol 8 bit	symbol 8 bit	bit length = N 8 bit	Encoded N bit
---------------------------	-----------------	-------------------------	------------------

위 사진과 같이 정리하였고 뒤로 이어지게 작성하였다.

```

// ===== Step 4. Encoding - Training Input & Calculate Cost ===== //
// Step 4.1 Encoding
input = fopen("training_input.txt", "r");
if (!input) {
    printf("file not exist\n");
    return 0;
}
tnum = 0;
chPrev = 0;
while (1)
{
    chTemp = fgetc(input);
    if (feof(input)) {
        TableControl(chPrev, end_of_data, tnum, 0); //end of data
        break;
    }
    TableControl(chPrev, chTemp, tnum, 0); //Encoding
    chPrev = chTemp;
}
fclose(input);

```

생성한 테이블을 바탕으로 Training Input을 인코딩한다.

```

// Control Using Normal/Adaptive
void TableControl(unsigned char prev, unsigned char cur, int &wr_tnum, int fname) {
    if (prev == 0) {
        GetFromTable(cur, wr_tnum, fname); // first character
    }
    else {
        if (GetFromAdaptiveTable(prev, cur, wr_tnum, fname) == 0) { //no adaptive table
            GetFromTable(cur, wr_tnum, fname); //use normal table
        }
    }
}

```

인코딩 시 Normal Huffman Table과 Context Adaptive Huffman Table을 선택하기 위한 알고리즘은 위와 같다.

우선 Adaptive Table에 preceding symbol에 대한 테이블이 존재하는지 살펴본다. GetFromAdaptiveTable의 함수가 0을 반환한다면 preceding symbol에 대한 테이블에 존재하지 않는다는 것을 뜻한다.

해당 문자에 대한 Adaptive Table이 존재하지 않는다면 Normal Table을 사용하여 인코딩한다. 위와 같이 코딩함으로써, 해당 값에 대해 일반화하여 parameter를 조절하여도 유동적으로 테이블 선택이 가능하다.

```
C = (double)data_byte / (double)original_byte; //압축 전 / 압축후
cost = C + (0.001)*(normal_byte + adaptive_byte); //cost 구하는 공식
// print in console
cout << "data byte: " << data_byte << endl;
cout << "original byte: " << original_byte << endl;
cout << "normal byte: " << normal_byte << endl;
cout << "adaptive byte: " << adaptive_byte << endl;
cout << "cost: " << cost << endl << endl;
```

이후 cost를 구하는 코드를 작성하여 parameter를 조절해보았다. Data byte는 압축된 결과를, original byte는 압축전의 파일을, normal byte는 허프만 테이블의 크기를, adaptive byte는 조건부 허프만 테이블의 크기를 뜻한다.

```
// ===== Step 5. Encoding - Test input 1, 2, 3 ===== //
// Step 5.1 encoding test1 file
FILE * fp_test1 = fopen("test_input1.txt", "r");
if (!fp_test1)
    printf("test_input1 안열림 \n");
tnum = 0;
chPrev = 0;
while (1){ ... }
fclose(fp_test1);
```

이후 위와 같이 test file1, 2, 3에 대해 압축을 모두 진행하였다.

여기까지가 Encoder.cpp의 결과이다. **EOD, Stuffing byte, byte align**을 고려한 내용은 **Result란에 결과화면과 같이 상세하게 작성하였다.**

decoder.cpp

```
int main() {
    // ===== Step 1. Create Table (vector) from hbs files ===== //
    vector<HuffmanTreeNode*> normal_table;
    MakeNormalTable(normal_table); //normal table

    vector<AdaptiveHuffmanTreeNode*> adaptive_table;
    MakeAdaptiveTable(adaptive_table); //adaptive table
}
```

디코더는 우선 두가지 종류의 바이너리 파일을 읽어 각각의 테이블을 vector형태로 생성한다.

```
int tnum = 0;
unsigned char preceding, preceding_hi, preceding_lo = NULL;
unsigned char ascii_table, ascii_hi, ascii_lo = NULL;
unsigned char bl_table, bl_hi, bl_lo = NULL;
unsigned char * codeword;
unsigned char last_bit_length = NULL;
while (1) {
    preceding_hi = fgetc(table_fp) << tnum; //preceding symbol읽어옴
    preceding_lo = fgetc(table_fp) >> (8 - tnum);
    if (feof(table_fp)) { //파일의 끝이면 끝
        break;
    }
    preceding = preceding_hi + preceding_lo; //preceding symbol
    fseek(table_fp, -1L, SEEK_CUR); // N = fp위치

    ascii_hi = fgetc(table_fp) << tnum;
    ascii_lo = fgetc(table_fp) >> (8 - tnum);
    ascii_table = ascii_hi + ascii_lo; //current symbol
    fseek(table_fp, -1L, SEEK_CUR); // N+1

    bl_hi = fgetc(table_fp) << tnum;
    bl_lo = fgetc(table_fp) >> (8 - tnum);
    bl_table = bl_hi + bl_lo; //bit length
    fseek(table_fp, -1L, SEEK_CUR); // N+2
}
```

```
int arr_size = (bl_table - 1) / 8 + 1; //codeword 저장할 곳
codeword = new unsigned char[arr_size];
memset(codeword, 0, arr_size * sizeof(unsigned char));
int code_cnt = bl_table;
for (int i = 0; i < arr_size; i++) {
    unsigned char temp_cw = NULL;
    if (tnum + code_cnt > 8) { //bit 가 빠져나간경우
        temp_cw = fgetc(table_fp) << tnum; //앞부분
        codeword[i] += temp_cw;
        temp_cw = fgetc(table_fp) >> (8 - tnum); //뒷부분
        code_cnt -= 8 - tnum;
        if (tnum - code_cnt < 0) { //뒤에 더 남은경우
            code_cnt -= tnum;
            codeword[i] += temp_cw;
            fseek(table_fp, -1L, SEEK_CUR);
        }
    } else { //뒤에 안남은경우
        last_bit_length = (8 - tnum) + code_cnt;
        unsigned char shamt = 8 - last_bit_length;
        temp_cw = (temp_cw & (0xff << shamt)); //다 왼쪽으로 채움
        codeword[i] += temp_cw;
    }
    else { //안빠져 나오는 경우
        temp_cw = fgetc(table_fp) << tnum;
        temp_cw = (temp_cw & (0xff << (8 - code_cnt))); //다 왼쪽으로 채움
        codeword[i] += temp_cw;
        last_bit_length = code_cnt;
    }
}
```

순서대로 읽어도 되는 이유는 중위 순회를 사용하여 인코딩 했기 때문에 vector에 넣어 index순서대로 읽어도 중위 순회와 같은 탐색이 이루어진다.

```
// ===== Step 2. Decoding training, test1,2,3 ===== //
unsigned char end_of_data = 0x7f;
int trainig_input = 0;
int test1 = 1;
int test2 = 2;
int test3 = 3;
DecodeControl(normal_table, adaptive_table, end_of_data, trainig_input); //training decode
DecodeControl(normal_table, adaptive_table, end_of_data, test1); //test1 decode
DecodeControl(normal_table, adaptive_table, end_of_data, test2); //test2 decode
DecodeControl(normal_table, adaptive_table, end_of_data, test3); //test3 decode
return 0;
```

이후 트레이닝, 테스트를 합쳐 총 4개의 파일에 대해 디코딩을 진행한다. **End of file**은 인코더, 디코더 모두 0x7f로 정했다.

Decode Control의 알고리즘은 다음과 같다.

```
string s, temp_s;
while (1) {
    code_hi = fgetc(code_fp) << tnum;
    code_lo = fgetc(code_fp) >> (8 - tnum);
    if (feof(code_fp)) {
        code_lo = 0;
    }
    fseek(code_fp, -1L, SEEK_CUR);
    code = code_hi + code_lo;
    for (int i = 7; i >= 0; i--) {
        temp_s = s + "";
        UCharToString(temp_s, code, 8 - i);
        if (first_char == true) { //데이터의 맨 처음일 경우
            decoded = SearchNormalTable(nv, temp_s);
            first_char = false;
        }
        else { //데이터의 맨 처음이 아닐 경우 normal vs adaptive
            if (IsExist(av, preceding_symbol)) { //preceding symbol에 대한 테이블이 존재하는 경우 사용
                decoded = SearchAdaptiveTable(av, preceding_symbol, temp_s);
            }
            else { //table이 존재하지 않는 경우 normal table 사용
                decoded = SearchNormalTable(nv, temp_s);
            }
        }
        if (decoded == eod) { //파일의 끝인 경우 함수를 종료시킨다.
            return;
        }
        if (decoded != NULL) { { ... } }
    }
    if (find_flag == false) { //string을 이어쓴다.
        s = temp_s;
        temp_s = "";
    }
    find_flag = false;
}
```

데이터의 맨 처음인 경우 Normal Table을 사용했기 때문에 디코딩에서도 Normal Table을 사용한다. 나머지의 경우 Adaptive와 Normal을 비교하여 사용한다. 우선 IsExist라는 함수를 만들어 사용했다. Adaptive table에 preceding symbol을 parameter로 전달하여 해당 preceding symbol에 대한 table이 존재하면 1을 나머지의 경우 0을 반환한다. 따라서 존재할 경우 Adaptive Table에서 인코딩 된 값을 비교하여 디코딩을 한다. 만약 IsExist가 0

을 반환한 경우 Adaptive Table이 존재하지 않는다는 의미 이므로 Normal Table을 이용하여 디코딩한다.

```
// 유효성 검사
bool IsExist(vector<AdaptiveHuffmanTreeNode*> &v, unsigned char prev) {
    int cnt = 0;
    while (1) {
        if (prev == v[cnt]->preceding_symbol) { //해당 심볼에 대한 테이블이 존재하면
            return true; //true반환
        }
        else {
            if (v.back() == v[cnt]) {
                break; //끝까지 못찾으면
            }
            cnt++;
        }
    }
    return false; //없다는 false반환
}
```

IsExist에 대한 함수는 위와 같다.

```
// Adaptive Table에서 값 찾는 함수
unsigned char SearchAdaptiveTable(vector<AdaptiveHuffmanTreeNode*> &v, unsigned char prev, string s) {
    int cnt = 0;
    while (1) {
        //preceding symbol, 찾는값 ch를 모두 만족하면
        if ((prev == v[cnt]->preceding_symbol) && (s == v[cnt]->encoded)) { //
            return v[cnt]->ascii_code; //해당 char값 반환 -> 디코딩 된값
        }
        else {
            if (v.back() == v[cnt]) {
                break; //끝까지 못찾으면
            }
            cnt++;
        }
    }
    return NULL; //NULL 반환
}
```

SearchAdaptiveTable은 위와 같다.

SearchNormalTable은 Assignment #4와 동일하다.


```

if (decoded != NULL) { //원하는 값을 찾은 경우
    preceding_symbol = decoded; //preceding symbol로 지정하고
    s = "";
    temp_s = "";
    find_flag = true; //flag들을 모두 원상복구 시킨다.
    WriteASCII(decoded, fname); //디코딩 한다.
    decoded = 0;
    tnum += (8 - i);
    fseek(code_fp, -1L, SEEK_CUR); // N -= 1
    while (1) {
        if (tnum >= 8) {
            fseek(code_fp, 1L, SEEK_CUR);
            tnum -= 8;
        }
        else break;
    }
    break;
}
}

```

위의 방법으로 원하는 decoded값을 찾게 되면 preceding symbol을 현재 값으로 설정하여 다음 문자에 대비하고, 모든 flag들을 초기화 한다. 또한 WriteASCII(Assignment #4와 동일)을 이용하여 디코딩 한 문자를 output으로 file에 출력한다.

3. Result

```
context_adaptive_huffman_table.hbs  X encoder.cpp
00000000 20 6E 05 01 02 20 38 42 05 70 70 A4 0E 00 C1 90 n... 8B.pp....
00000010 34 02 09 03 08 19 20 69 04 42 06 F0 45 20 6B 07 4..... i.B..E k.
00000020 60 40 50 0E C4 81 50 1D 91 03 C8 3B 32 06 C0 56 `@P...P....;2..V
00000030 90 33 02 B8 81 B4 15 E4 0C 00 D0 10 39 03 42 40 .3.....9.B@
00000040 A4 11 10 40 8A 15 12 10 28 05 44 A4 0B 41 51 31 ..@....( .D..AQ1
00000050 02 18 54 4E 40 EC 0F 14 81 04 1A 32 04 90 69 08 ..TN@.....2..i.
00000060 19 C1 A5 20 65 06 98 81 34 26 70 40 E2 15 39 10 ... e...4&p@..9.
00000070 1F 89 CE 60 02 02 A1 39 CC 02 40 64 27 39 80 88 ...`...9...@d'9..
00000080 0C 04 E7 30 19 00 80 A4 E6 04 10 0A 8A 4E 60 49 ...0.....N`I
00000090 02 00 A4 E6 05 10 17 0A 4E 60 59 00 F0 A4 E6 06 .....N`Y.....
000000a0 10 0B 0A 4E 60 69 01 08 A4 E6 07 10 06 8A 4E 60 ...N`i.....N`
000000b0 79 00 48 9C E6 08 20 25 13 9C C1 24 04 62 73 98 y.H... %...$.bs.
000000c0 28 80 98 4E 73 05 90 3D 89 4E 60 C4 0F A2 53 98 (.Ns...=.N`...S.
000000d0 39 00 60 9C E6 10 20 17 13 9C C2 24 03 82 73 98 9.`... ..$.s.
000000e0 48 80 90 4E 73 09 90 03 89 CE 61 42 00 B1 39 CC H..Ns....aB..9.
000000f0 2A 40 26 27 39 85 88 06 C4 E7 30 B9 00 30 9C E6 *@&'9....0..0..
00000100 18 20 08 13 9C C3 24 00 A2 73 98 68 80 10 4E 73 .....$.s.h...Ns
00000110 0D 90 0C 09 CE 61 C2 01 41 39 CC 3A 40 1E 27 39 ...a..A9.:@.'9
00000120 87 88 04 44 E7 30 F9 02 C8 94 E6 20 40 BA 25 39 ...D.0....@.%9
00000130 88 90 3F 09 4E 62 44 0F 82 53 98 99 00 D0 94 E6 ..?.NbD..S.....
00000140 28 40 3A 25 39 8A 90 17 89 4E 62 C4 07 A2 53 98 (@:%9....Nb...S.
00000150 B9 01 60 94 E6 30 40 84 25 39 8C 90 1F 09 4E 63 ...0@.%9....Nc
00000160 44 06 22 53 98 D9 01 68 94 E6 38 40 66 25 39 8E D."S...h..8@f%9.
00000170 90 1A 89 4E 63 C4 07 02 53 98 F9 02 A8 8C E6 40 ...Nc...S.....@
00000180 81 2C 46 73 22 40 F4 23 39 92 20 78 11 9C C9 90 ..Fs"@.#9. x...
00000190 25 08 CE 65 08 17 04 67 32 A4 07 42 33 99 62 02 %..e...g2..B3.b.
000001a0 91 19 CC B9 01 B8 8C E6 60 80 AC 46 73 32 40 68 .....Fs2@h
000001b0 23 39 9A 20 36 11 9C CD 90 28 88 CE 67 08 17 84 #9. 6....(.g...
000001c0 67 33 A4 0B E2 33 99 E2 05 B1 19 CC F9 00 50 84 g3...3.....P.
000001d0 E6 81 00 90 84 E6 89 00 70 84 E6 91 01 10 84 E6 .....p.....
000001e0 99 00 C8 84 E6 A1 00 F8 84 E6 A9 00 08 84 E6 B9 .....
000001f0 01 38 84 E6 C1 01 D8 84 E6 C9 01 C8 84 E6 D1 01 .8.....
00000200 E0 84 E6 D9 03 F8 84 E6 E1 00 10 8C E6 E8 80 0C .....
00000210 46 73 76 40 B0 21 39 BC 40 8E 21 39 BE 40 8C 17 Fsv@.!9.@.!9.@..
00000220 39 C8 14 C2 27 48 13 02 27 88 1A 82 27 C8 1D 00 9....H..!.....
00000230 50 80 10 05 00 10 70 40 5F 00 11 05 00 00 40 .....
```

위 그림은 Context Adaptive Huffman Table을 바이너리 편집기로 열어서 확인한 모습이다.

해당 테이블과 normal 허프만 테이블을 사용하여 인코딩 된 training 파일은 다음과 같다.

and Alice's first thought was that it might belong to one of the
doors of the hall; but, alas! either the locks were too large, or
the key was too small, but at any rate it would not open any of
them. However, on the second time round, she came upon a low
curtain she had not noticed before, and behind it was a little
door about fifteen inches high: she tried the little golden key
in the lock, and to her great delight it fitted!

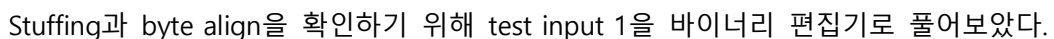
트레이닝의 마지막 문자는 ! -> EOD순서로 오게 된다.

조건부 허프만은 ASCII 32일때만 생성이 되었으므로 normal Huffman table을 사용하여

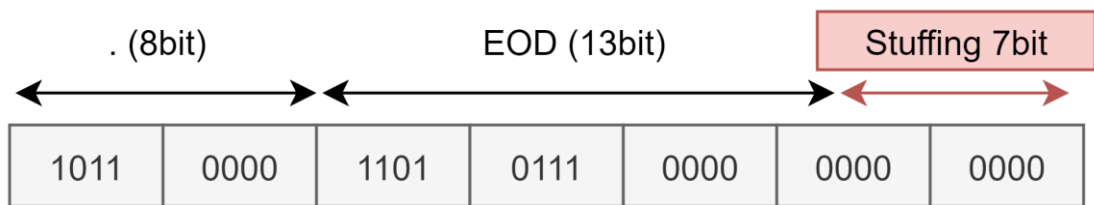
느낌표는 0110 1100

으로 인코딩 되었다.

따라서 마지막의 8D 9A E0을 binary로 풀어보았다.



은 1011 0000 으로 인코딩 된다.



따라서 위와 같이 "."에 대해 8bit, EOD에 대해 13bit, 하고 마지막 7bit가 남는다. 이를 메꾸기 위해 7bit를 stuffing bit 0으로 채워주게 되고 이를 byte align이라고 한다. Unsigned char를 읽기 위해서는 8bit씩 파일이 존재해야 하기 때문이다.

따라서 **Byte Align**과 **Stuffing bit**, **EOD**에 대해 모두 정확히 작동함을 확인했다.

$Cost = C + 0.0001 * N_table$ ($C = \text{압축전 file bytes} / \text{압축후 file bytes}$)

($N_table = \text{Huffman_table.hbs} + \text{context_adpative_huffman_table.hbs bytes}$)

Cost를 구하기 위해 Frequency에 따라 Cost가 어떻게 변하는지 실험해보았다.

1) Frequency > 1 cost: 3.8499

```
data byte: 2758
original byte: 6551
normal byte: 484
adaptive byte: 33805
cost: 3.8499
```

Data byte는 인코딩 된 결과를 의미하며 2758 byte로 42%의 압축률을 보인다.

하지만 조건부 허프만 테이블의 크기가 너무 커서 cost가 너무 높게 측정된 것을 확인할 수 있다.

2) Frequency > 10 cost: 2.63872

```
data byte: 2821
original byte: 6551
normal byte: 482
adaptive byte: 21599
cost: 2.63872
```

압축률은 42%에서 43%로 증가하였지만 조건부 허프만 테이블의 크기가 여전히 너무 커서 코스트가 높게 계산되었음을 확인했다.

3) Frequency > 100 cost: 1.48602

```
data byte: 3030
original byte: 6551
normal byte: 483
adaptive byte: 9752
cost: 1.48602
```

압축률은 43%에서 46%로 증가하였다. 하지만 조건부 허프만 테이블의 크기가 많이 감소하여 cost가 많이 작아짐을 확인했다.

4) Frequency > 200 cost: 1.25084

```
data byte: 3093
original byte: 6551
normal byte: 482
adaptive byte: 7305
cost: 1.25084
```

압축률은 46%에서 47%로 증가하였고 cost는 여전히 감소 추세이다.

5) Frequency > 300 cost: 0.99476

```
data byte: 3232
original byte: 6551
normal byte: 488
adaptive byte: 4526
cost: 0.99476
```

데이터의 특성상 frequency가 300이 넘는 것이 많이 존재하지 않아 테이블의 크기가 많이 줄었다. 압축률은 49%정도이다. Cost가 200초과에 비해 많이 줄어든 것을 확인했다.

6) Frequency > 350 cost: 0.885662

```
data byte: 3346
original byte: 6551
normal byte: 482
adaptive byte: 3267
cost: 0.885662
```

계속 늘릴수록 cost는 줄어드는 것을 확인할 수 있다. 약 51%의 압축률을 갖는다.

7) Frequency > 400 cost: 0.754921

```
data byte: 3444
original byte: 6551
normal byte: 482
adaptive byte: 1810
cost: 0.754921
```

압축률은 52%이고 코스트는 계속 줄어드는 것을 확인했다.

8) Frequency > 500 cost: 0.70359

```
data byte: 3489
original byte: 6551
normal byte: 482
adaptive byte: 1228
cost: 0.70359
```

압축률은 53%이고 코스트는 계속 줄어든다.

9) Frequency > 600 cost: 0.647449

```
data byte: 3530
original byte: 6551
normal byte: 483
adaptive byte: 603
cost: 0.647449
```

압축률은 53.8%이고 코스트는 1.61로 매우 현저히 줄어드는 것을 확인했다. 이번 과제
의 경우, 600보다 빈도수가 많은 경우는 Adaptive Huffman Table이 ASCII가 32인경
우 1개만 생성이 된다.

Frequency > 600을 채택하여 test를 돌린 결과

training_input 의 압축 전 -> 압축 후 -> 디코딩 후 결과


training_input 속성

일반

보안

자세히

이전 버전




training_input

파일 형식:

텍스트 문서(.txt)

연결 프로그램:

 메모장

위치:

C:\Users\nyc09\source

크기:

6.39KB (6,551 바이트)

디스크 할당 크기:

8.00KB (8,192 바이트)


training_input_code 속성

일반

보안

자세히

이전 버전




training_input_code

파일 형식:

HBS 파일(.hbs)

연결 프로그램:

 메모장

위치:

C:\Users\nyc09\source

크기:

3.44KB (3,529 바이트)

디스크 할당 크기:

4.00KB (4,096 바이트)


training_output 속성

일반

보안

자세히

이전 버전




training_output

파일 형식:

텍스트 문서(.txt)

연결 프로그램:

 메모장

위치:

C:\Users\nyc09\source

크기:

6.39KB (6,551 바이트)

디스크 할당 크기:

8.00KB (8,192 바이트)

약 53.8%의 압축률을 보이며 정상적으로 인코딩 디코딩이 되었음을 확인했다.

test input 1의 압축 전 -> 압축 후 -> 디코딩 후 결과

test_input1 속성

일반

보안

자세히

이전 버전

test_input1

파일 형식: 텍스트 문서(.txt)

연결 프로그램:  메모장

위치: C:\Users\wnyc09\source

크기: 6.68KB (6,850 바이트)

디스크 할당 크기: 8.00KB (8,192 바이트)

test_input1_code 속성

일반

보안

자세히

이전 버전

test_input1_code

파일 형식: HBS 파일(.hbs)

연결 프로그램:  메모장

위치: C:\Users\wnyc09\source

크기: 3.63KB (3,718 바이트)

디스크 할당 크기: 4.00KB (4,096 바이트)

test_output1 속성

일반

보안

자세히

이전 버전

test_output1

파일 형식: 텍스트 문서(.txt)

연결 프로그램:  메모장

위치: C:\Users\wnyc09\source

크기: 6.68KB (6,850 바이트)

디스크 할당 크기: 8.00KB (8,192 바이트)

약 54.2%의 압축률을 보인다.

test 2 압축 전 -> 압축 후 -> 디코딩 후

test_input2 속성

일반

보안

자세히

이전 버전



test_input2

파일 형식:

텍스트 문서(.txt)

연결 프로그램:

 메모장

위치:

C:\Users\Wnyc09\source

크기:

7.26KB (7,444 바이트)

디스크 할당 크기:

8.00KB (8,192 바이트)

test_input2_code 속성

일반

보안

자세히

이전 버전



test_input2_code

파일 형식:

HBS 파일(.hbs)

연결 프로그램:

 메모장

위치:

C:\Users\Wnyc09\source\W

크기:

3.96KB (4,063 바이트)

디스크 할당 크기:

4.00KB (4,096 바이트)

test_output2 속성

일반

보안

자세히

이전 버전



test_output2

파일 형식:

텍스트 문서(.txt)

연결 프로그램:

 메모장

위치:

C:\Users\Wnyc09\source

크기:

7.26KB (7,444 바이트)

디스크 할당 크기:

8.00KB (8,192 바이트)

압축률은 54.5%정도이고 정상적으로 인코딩, 디코딩 되었음을 확인했다.

test 3의 압축 전 -> 압축 후 -> 디코딩의 결과이다.


test_input3 속성

일반

보안

자세히

이전 버전




test_input3

파일 형식:

텍스트 문서(.txt)

연결 프로그램:

 메모장

위치:

C:\Users\Wnyc09\source

크기:

5.95KB (6,094 바이트)

디스크 할당 크기:

8.00KB (8,192 바이트)

test_input3_code 속성

일반

보안

자세히

이전 버전




test_input3_code

파일 형식:

HBS 파일(.hbs)

연결 프로그램:

 메모장

위치:

C:\Users\Wnyc09\source

크기:

3.33KB (3,419 바이트)

디스크 할당 크기:

4.00KB (4,096 바이트)


test_output3 속성

일반

보안

자세히

이전 버전



test_output3

파일 형식:

텍스트 문서(.txt)

연결 프로그램:

 메모장

위치:

C:\Users\Wnyc09\source

크기:

5.95KB (6,094 바이트)

디스크 할당 크기:

8.00KB (8,192 바이트)

압축률은 약 56.1% 정도이고 정상적으로 작동함을 확인했다.

Training : 53.8%

Test1 : 54.2%

Test2 : 54.5%

Test3 : 56.1%

로 크게 차이 나지 않음을 확인했고, 대략 54~55%정도의 압축률을 갖는 것을 알 수 있다.

Huffman_table.hbs 의 크기

Huffman_table 속성

일반	보안	자세히	이전 버전
<div> Huffman_table</div>			
<hr/>			
파일 형식:	HBS 파일(.hbs)		
연결 프로그램:	 메모장		
<hr/>			
위치:	C:\Users\Wnyc09\source\W		
크기:	485바이트 (485 바이트)		
디스크 할당 크기:	0바이트		

Assignment #4에서는 모든 파일마다 각각 인코딩 디코딩을 진행하여 빈도 수가 0 인 문자에 대해서 테이블을 생성하지 않아 크기가 작았지만 본 과제의 경우 128개의 ASCII모두 생성하기 때문에 약간 크기가 늘어난 것을 확인했다.

context_adaptive_huffman_table.hbs 크기

context_adaptive_huffman_table 속성

일반	보안	자세히	이전 버전
<div> context_adaptive_huffman_</div>			
<hr/>			
파일 형식:	HBS 파일(.hbs)		
연결 프로그램:	 메모장		
<hr/>			
위치:	C:\Users\Wnyc09\source\W		
크기:	603바이트 (603 바이트)		
디스크 할당 크기:	0바이트		

Frequency > 600으로 진행하여 ASCII " " (빈도수 1182)에 대해서만 테이블이 생성되어 603바이트의 크기를 갖는 것을 확인했다.

4. Consideration

이번 과제에서 Assignment #4을 확장하여 조건부 허프만 코딩을 하여 어떤 효과가 생기는지, 압축률은 어떻게 변하는지에 대한 것을 알아보았다.

본 과제에서 실험 결과는 context_adaptive_huffman_table.hbs가 생성되고 사용됨을 보여야 하기 때문에 빈도수가 600이 넘는 경우로 설정하여 결과를 측정했다. Assignment #4와 비교하면 context_adaptive_huffman_table을 사용하면 압축률이 더 좋은 것으로 확인되었다.

과제를 하기 전에 training_input에 딱 맞게 encoded를 하면 test의 경우 overfitting에 의해 압축률이 저조할 것으로 예상했다. 하지만 본 과제에서 제공된 training_input이 크기가 충분하고 다양한 경우의 수를 내포하고 있었기에 1~2%의 압축률만 차이가 날 뿐 큰 차이가 없음을 확인했다. 더욱 general한 결과를 갖고 싶다면 training_input의 크기를 키워 표본이 커지는 효과를 줄 수 있을 것이다.

과제를 하기 전에는 빈도수에 따라 Table의 크기와 data byte간의 압축률에 따라 최적의 parameter가 있을 것으로 예상했다. 그러나 실제로 코드를 작성하여 실험을 진행한 결과 예상과 다르게 normal table만을 사용하는 것이 가장 cost가 좋게 측정되었다. 따라서 이에 대해 원인을 추측해 보았다. 람다(Lagrange Multiplier)값이 0.001도 너무 크게 책정이 되었던 것 같다. 0.0001로 바뀌서 Huffman table의 의존도가 낮아져 cost가 제대로 측정된 것 같다.

본 프로젝트에서 각각의 파일에 대해가 아니라 전체적으로 고려해 본 결과는 다음과 같다.

Test1 + test2 + test3 압축 전: 20388 바이트

Test1 + test2 + test3 압축 후: 11200 바이트

Normal + adaptive: 1088 바이트

$20388 - 11200 - 1088 = 8100$ 바이트로 총 8100 바이트가 줄어들었다. 여기서 주목할 점은 table은 계속 재사용이 되기 때문에 입력하는 input의 개수, 크기가 많아지고 커질수록 context adaptive Huffman table의 효율이 좋아진다는 것을 유추할 수 있다.

만약 데이터의 단위가 바이트가 아니라 KB MB GB라면 분명 notable한 차이를 보일 것으로 추정된다. 따라서 cost를 구할 때 training으로만 구할 것이 아니라 test input까지 모두 고려하여 percentage만 구하는 것이 아니라 실제로 얼마나 save되는지 구해보는 것이 중요할 것 같다. 이에 따라 매 경우마다 context adaptive table의 개수를 정하는 값은 달라질 것이다. 본 과제에서는 normal Huffman table만 사용하여 압축하는 것이 가장 좋은 결과를 도출함을 알 수 있다.