알고리즘 설계와 분석

(Design and Analysis of Algorithms)

Machine Problem 3

20161603
신민준

Table of Contents

1 Implementation

```
/**
 * @brief Class for implementing Huffman Code Encryption & Decryption
 */
class Huffman {
  protected:
    std::string inFilename, outFilename;
    std::ifstream inFile;
    std::ofstream outFile;
    HuffmanOption option;
    node_ptr huffmanTree;
    std::vector<node_ptr> huffmanCode;

    /* compare class for priority queue */
    class compare {
      public:
        bool operator()(const node_ptr& v, const node_ptr& u) const {
          return v->freq > u->freq;
        }
    };
    priority_queue<node_ptr, std::vector<node_ptr>, compare> pq;

    void makeHuffmanTree(void);
    void assignBinaryCode(node_ptr, std::string);
    void initialize(void);
    void openFile(void);
    void writeToFile();
    void reconstructTree(void);
    char binToByte(string);
    string byteToBin(unsigned char*, int);
    void addToTree(node_ptr);

  public:
    Huffman(std::string, HuffmanOption);
    const string getHuffmanCodeOf(char);
    void encode(void);
    void decode(void);
};
```

Code 1

[Code 1]에서는 파일 압축을 위해 Huffman code를 생성하고, 생성한 코드에 따라 파일을 압축 / 압축해제하는 클래스 Huffman을 보이고 있습니다.

Huffman code를 생성하기에 앞서, 아래의 [Code 2]와 같이 makeHuffmanTree 메소드를 사용해 Huffman tree를 구조하도록 했습니다.

```
/**
 * Create a huffman tree from given data.
 */
void Huffman::makeHuffmanTree(void) {
   int occurence[ASCII_MAX_VALUE+1] = {0};
   int len;
   char* buffer;

   if(!inFile.is_open())
      openFile();

   /* read whole file */
   inFile.seekg(0, ios::end);
   len = inFile.tellg();
   inFile.seekg(0, ios::beg);

   buffer = new char[len];

   inFile.read(buffer, len);

   /* record occurences of ASCII chars in file */
   for(int i=0 ; i<len ; i++) {
      occurence[(unsigned int)buffer[i]]++;
   }
   delete[] buffer;

   /* create priority queue from recorded characters */
   for(int i=0 ; i<ASCII_MAX_VALUE+1 ; i++) {
      if(!occurence[i])
         continue;
      node_ptr v = new _huffman_node();
      v->id = i;
      v->freq = occurence[i];
      pq.push(v);
      huffmanCode.push_back(v);
   }
```

Code 2

먼저, 이 메소드는 입력 파일의 내용을 전부 다 읽고 해당 파일에서 등장한 모든 문자에 대해서 각각 문자들의 빈도수를 기록하고, 빈도수를 기준으로 내림차순으로 정렬하는 priority

queue를 사용해 각 문자들을 노드 구조체 _huffman_node에 저장합니다. 해당 노드 구조체는 다음과 같이 구현했습니다.

```
struct _huffman_node {
   char id;
   int freq;
   std::string code;
   _huffman_node* left;
   _huffman_node* right;

   /* initialise constructor */
   _huffman_node() {
      this->left = this->right = NULL;
      this->id = INVALID_CHAR;
   }
};
typedef _huffman_node* node_ptr;
```

Code 3

```
   while(pq.size() > 1) {

      /* popping smallest & 2nd-smallest freq from priority queue */
      node_ptr v = pq.top();
      pq.pop();
      node_ptr u = pq.top();
      pq.pop();

      /* union two popped node, then add it to the priority queue again */
      node_ptr newNode = new _huffman_node();
      newNode->left = v;
      newNode->right = u;
      newNode->freq = v->freq + u->freq;
      pq.push(newNode);
   }

   /* huffman tree is stored in PQ's top */
   this->huffmanTree = pq.top();
   pq.pop();
}
```

Code 4

이후, priority queue로부터 가장 적은 빈도를 가지는 노드 2개를 서로 union하고, 새로 만들

어진 이 노드를 priority queue로 다시 추가하는 [Code 3]의 과정을 반복해 Huffman tree를 완성하게 됩니다.

Huffman tree가 완성되면, tree에서 문자들에 해당하는 각 leaf node들까지 가는 경로를 0과 1로 표현해준 것이 해당 문자의 Huffman code가 됩니다. 따라서, 해당 부분은 다음 [Code 4]와 같이 재귀적으로 구현했습니다.

```
/**
 * @brief Assign binary code to huffman tree, recursively
 *
 * @param root Root node of huffman tree
 * @param code Current bitstring. Do assign "" when calling.
 */
void Huffman::assignBinaryCode(node_ptr root, string code) {
   if(!root) {
      cerr << "DFS error!" << endl;
      exit(1);
   }
   if(root->left == NULL and root->right == NULL) {
      root->code = code;
      return;
   }
   assignBinaryCode(root->left, code + "0");
   assignBinaryCode(root->right, code + "1");
}
```

Code 5

위 assignBinaryCode 메소드는 특정 노드 기준, 왼쪽 child로 탐색을 이어가면 코드에 0을, 오른쪽 child로 탐색을 이어가면 1을 추가해 가면서 마지막 leaf node에 다다르면 생성된 코드를 해당 노드에 추가해주는 방식으로 구현했습니다.

위의 과정을 거치면, 입력 파일에 존재하는 모든 character마다 하나씩의 Huffman code를 갖게 됩니다. 다만, 입력 파일을 그대로 압축해 출력 파일에 쓰게 되면, 압축을 해제할 때 어떤 character가 어떤 Huffman code와 mapping 되어있는지 알 수 없으므로, character와 huffman code 간의 mapping 정보인 metadata들이 "Header" 형식으로 압축 파일의 처음 bit들에 오도록 압축 파일을 설계했습니다. 의도한 파일의 구조는 다음 그림과 같습니다.

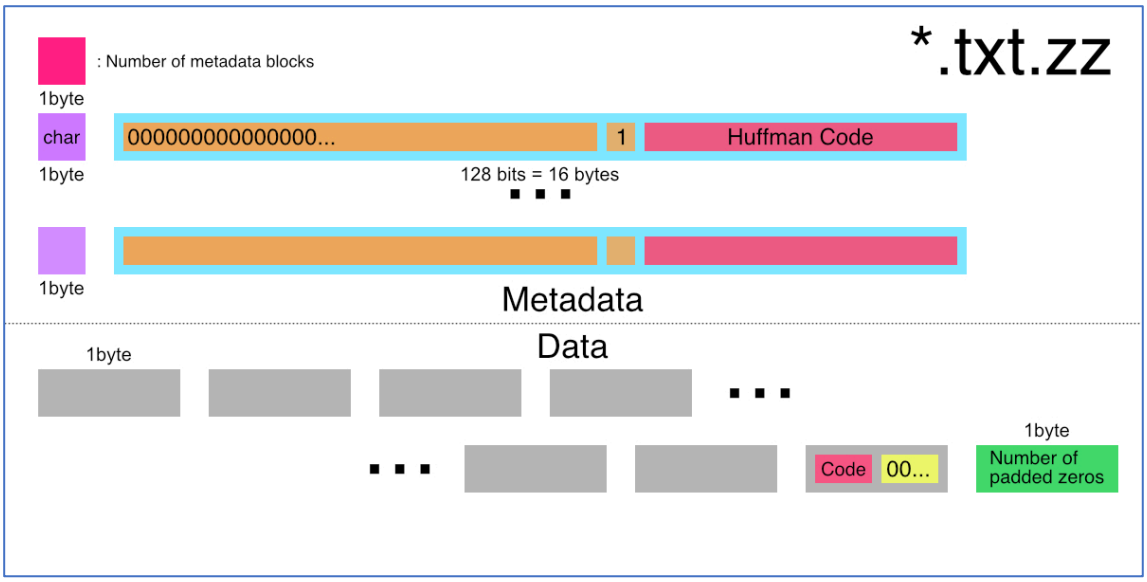

Figure 1

위 다이어그램에서 볼 수 있듯이, 메타데이터 블록들의 수, 즉 character-code 쌍의 mapping 개수를 첫 byte 값에 쓰고, 이후 메타데이터 블록들이 쓰여집니다. 메타데이터 블록은 총 17 바이트로, 첫 바이트에는 mapping에서의 character 값이 쓰이고, 나머지 16바이트(=128 비트)에는 Huffman code가 0으로 padding 되어있는 형태로 저장됩니다. 이 때, Huffman code의 내용이 시작하기 직전의 bit를 1로 두어 메타데이터를 읽을 때 코드 값을 잘못 읽는 것을 방지했습니다.

메타데이터를 다 읽고 나면, 바로 Huffman code로 압축된 데이터들이 쓰여지도록 했습니다. 이 때, 총 인코딩 된 bit string의 길이가 정확하게 8로 나누어떨어지지 않을 수 있기 때문에, 마지막 남는 bit들은 뒤에 0 padding을 추가적으로 붙여 하나의 byte로 만들고, 파일의 마지막 바이트에는 추가적으로 붙인 0 padding의 수를 쓰도록 했습니다.

따라서, 이와 같이 압축된 파일을 다시 압축 해제하기 위해서는, 1) 먼저 첫 바이트를 읽고, 2) 읽은 바이트의 값 만큼의 메타데이터를 읽어와 character-code 매핑을 복구한 뒤, 3) 마지막 byte를 읽어 데이터의 마지막 0 padding의 수를 알아내고, 4) 0 padding을 제외한 Data 부분을 bit string으로 읽으면서 character-code mapping으로부터 original text를 복구하면 됩니다. 이 과정은 각각 다음의 reconstructTree 메소드와 decode 메소드로 구현되어 있습니다.

```
/**
 * @brief Reconstruct tree based on read bit string header.
 */
void Huffman::reconstructTree(void) {
   int nodeCount, padCount;
   unsigned char buffer[CODEBLOCK_SIZE];
   node_ptr temp;
   nodeCount = inFile.get();

   /* initialize root of huffman tree */
   this->huffmanTree = new _huffman_node();

   /* read headers */
   for(int i=0 ; i<nodeCount ; i++) {
      temp = new _huffman_node();

      /* get a id byte from file */
      temp->id = inFile.get();

      /* read 128-bit(16Byte) block containing code */
      inFile.read((char*)buffer, CODEBLOCK_SIZE);

      string codeBit = byteToBin(buffer, CODEBLOCK_SIZE);

      /* count number of 0 paddings in code bits */
      for(padCount = 0 ; codeBit[padCount] == '0' ; padCount++);

      /* remove 0 paddings and leading 1 from code bits */
      codeBit = codeBit.substr(padCount+1);
      temp->code = codeBit;

      /* add new node to code list, and add to tree */
      this->huffmanCode.push_back(temp);
      addToTree(temp);
   }
   inFile.close();
}

/**
 * @brief Decode input file's contents, and writes decoded text to file
 */
void Huffman::decode() {
   outFile.open(outFilename, ios::out);
   inFile.open(inFilename, ios::in|ios::binary);
   unsigned char mapSize;
   unsigned char trailing0s;
   unsigned char temp;
   vector<unsigned char> text;
   string decoded;
   string binary;

   mapSize = inFile.get();

   /* go to the end of the file, get 1 byte: number of trailing zeros */
   inFile.seekg(-1, ios::end);
   trailing0s = inFile.get();
```

```
   /* get to the start of bit stream */
   inFile.seekg(1 + (CODEBLOCK_SIZE+1)*mapSize, ios::beg);

   temp = inFile.get();
   while(inFile.good()){
      text.push_back(temp);
      temp = inFile.get();
   }
   text.pop_back(); // remove trailing 1 byte: number of trailing zeros

   binary += byteToBin(&text[0], text.size());
   binary = binary.substr(0, binary.size()-trailing0s);

   node_ptr now = this->huffmanTree;
   for(std::vector<unsigned char>::size_type i=0 ; i<binary.size() ; i++) {
      if(now->left == NULL and now->right == NULL) {
         decoded += now->id;
         now = this->huffmanTree;
      }
      if(binary[i] == '0') {
         now = now->left;
      } else {
         now = now->right;
      }
   }

   outFile.write(decoded.c_str(), decoded.size());
   outFile.close();
   inFile.close();
}
```

Code 6

2 Experiment

2.1 Environment Specifications

MacBook Pro (13-inch, 2018)

OS - macOS Catalina version 10.15.1

CPU - 2.3 GHz Intel Core i5

RAM - 8GB 2133 MHz LPDDR3

Graphics - Intel Iris Plus Graphics 655 1538MB

Shell - zsh 5.7.1 (x86_64-apple-darwin18.2.0)

Compiler - Apple clang version 11.0.0 (clang-1100.0.33.12)

2.2 Experiment Setup

Makefile에서 g++ 컴파일러와 C++ 14 standard를 사용하도록 설정했습니다. Makefile의 내용은 다음과 같습니다.

```
# Compiler: g++
CXX = g++
# Flags: C++14 standard
CPPFLAGS += -std=c++14
# Libmath
#LIBS += -lm

# making the executable
mp3_20161603: mp3_20161603.cpp
   $(CXX) $(CPPFLAGS) mp3_20161603.cpp -o mp3_20161603 $(LIBS)

clean:
   rm mp3_20161603
```

Figure 2 Makefile

실험에 사용할 input은 500byte, 60KB, 900KB, 12MB, 총 4개의 각기 다른 크기의 텍스트 파일을 선택했습니다. 파일의 내용은 다음처럼 일반적인 Lorem ipsum 텍스트입니다.

Figure 3

추가적으로, worst-case input을 테스트 하기 위해 다음과 같은 터미널 명령어로 worst-case 텍스트 파일을 생성했습니다.

```
python3 -c 'for x in range(0, 128): print(chr(x)*1000, end="")' > all.txt
```

Code 7

2.3 Result Data

Lorem Ipsum 텍스트 파일에 대한 결과는 다음과 같았습니다.

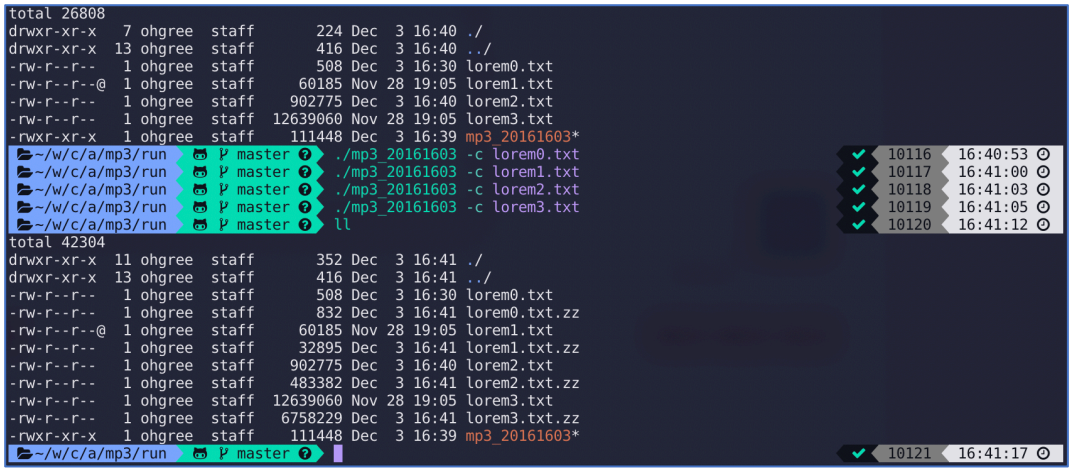

Figure 4

위 결과에 대한 압축률을 정리해 표로 나타내면 다음과 같습니다.

	lorem0.txt	lorem1.txt	lorem2.txt	lorem3.txt
raw	508 Bytes	60185 Bytes	902775 Bytes	12639060 Bytes
compressed	832 Bytes	32895 Bytes	483382 Bytes	6758229 Bytes
ratio	164%	55%	54%	53%

Table 1

다음은 worst-case input에 대한 압축 결과입니다.

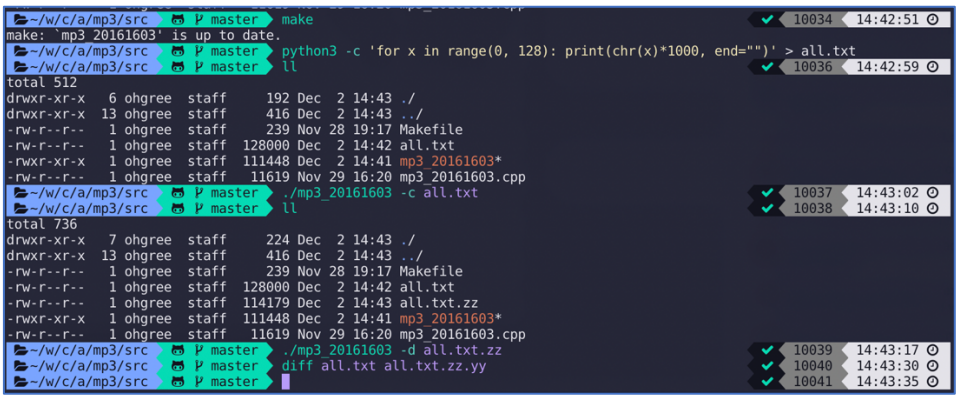

Figure 5

위 결과를 표로 나타내면 다음과 같습니다.

	worst case txt
raw	128000 Bytes
compressed	114179 Bytes
ratio	89%

Table 2

2.4 Data Analysis

2.4.1 Random text file – Lorem Ipsum

Lorem Ipsum은 일반적으로 사용하는 문장과 가장 많이 닮아있어, 대부분의 보편적인 텍스트 문서와 사용되는 문자가 비슷하게 구성됩니다. 따라서, 실생활에서 저장하는 텍스트와 가장 유사한 형태의 테스트 케이스라고 볼 수 있는데, [Table 1]의 결과에서 볼 수 있듯이, 대부분의 경우에서 50% 정도의 압축률을 보이고 있습니다. 그러나, 파일의 크기가 매우 작은 lorem0.txt 파일의 경우, 압축 이후 파일의 크기가 오히려 커진 것을 확인할 수 있는데, 이는 메타데이터를 저장하는 헤더의 크기가 고정되어있기 때문이라고 볼 수 있습니다. 실제로 [Table 1]의 데이터를 표로 표현하면, 아래의 [Figure 6]와 같은 모습의 차트를 확인할 수 있습니다.

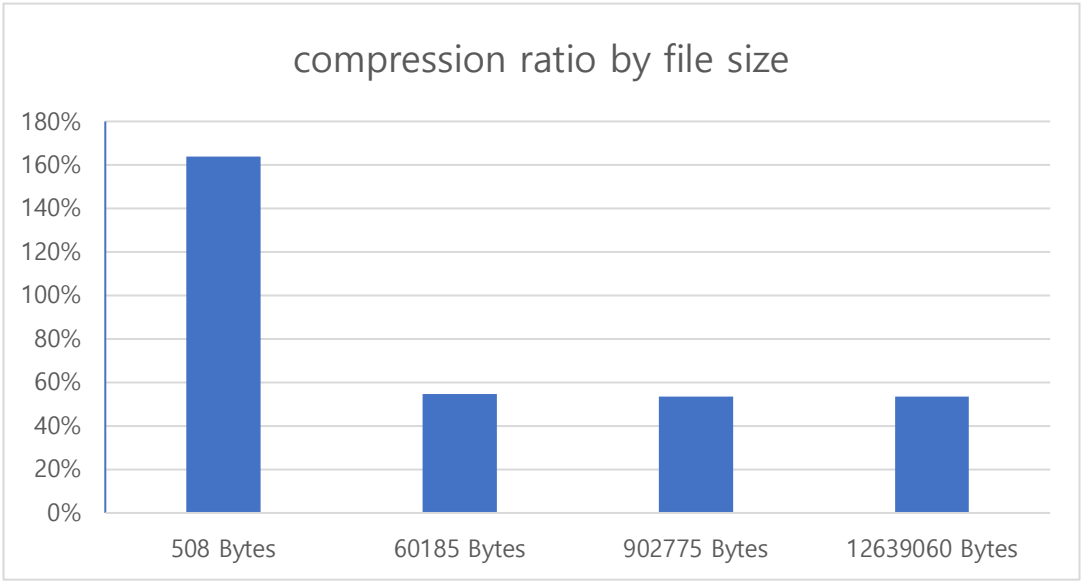

Figure 6

508 바이트 수준의 극도로 작은 크기의 파일에서는 압축률 손실이 있으나, 파일의 사이즈가

커지면 메타데이터의 크기가 파일 전체의 데이터 크기에 비해 무시할 수 있을 정도로 작아져, 특정 파일 사이즈 이후부터는 일정한 압축률을 유지하는 것을 볼 수 있기에, 압축 구현이 정확하게 이루어졌다고 평가할 수 있습니다.

2.4.2 Worse-case text file

Huffman Tree에서, 낮은 빈도를 가지는 문자일수록 더 긴 bit string code와 mapping 되므로, 압축률의 측면에서 볼 때의 worst case를 보이는 input file은 모든 문자가 같은 개수만큼 들어있는 파일입니다.

이 프로그램은 non-extended ASCII code만을 지원하므로, 0~127의 값을 가지는 총 128개의 문자만을 지원합니다. 따라서, 각 128개의 문자들을 같은 빈도로 쓴 worst case 텍스트 파일을 생성해 실험을 진행했습니다. 추가적으로 이 때, 메타데이터의 크기가 결과값에 주는 영향을 최소화하기 위해 충분히 큰 크기의 텍스트 파일이 되도록 파일을 생성했습니다.

[Table 2]에서 볼 수 있듯, 이 실험의 결과는 Lorem ipsum 실험에서 일반적인 텍스트 파일로 실험한 결과인 50% 대의 압축률에 비교하면 안좋은 수준의 압축률을 보였지만, 결론적으로는 원본 파일보다 10% 정도 더 작은 압축된 파일을 만들어 내었습니다.

이러한 worst case류의 파일들은 매우 드물다는 점을 감안하면 이 프로그램은 충분한 수준의 압축률을 지녔다고 할 수 있습니다.

3 Additional Comments

- 작은 크기의 파일을 압축할 때, 메타데이터를 저장하는 부분이 차지하는 비중이 상대적으로 커져 압축 효율에 부정적인 영향을 끼치는 것을 확인할 수 있었습니다. 만약 메타데이터의 구조를 조금 더 효율적으로 바꿔, 메타데이터에 대한 메타데이터를 추가적으로 작성해 작은 파일에 대한 압축률을 높일 수 있을 것입니다. 예를 들어, 현재 사용하고 있는 메타데이터 형식은 Huffman code 를 16 바이트의 고정된 크기로 기록하는데, 이를 가변적으로 설정해 크기를 줄일 수 있습니다. 파일의 첫 바이트로 메타데이터 블록의 수와 더불어 Huffman code 의 바이트 수도 추가적으로 명시하도록

구현하면, 불필요한 zero padding 을 최소화 해 전체 파일 사이즈가 줄어들 것으로 기대할 수 있습니다.