2018년 2학기 알고리즘 설계와 분석
CSE3081-01반 MP3 과제 보고서

학번: 20171665
이름: 이 선 호

2018. 12. 12

목 차

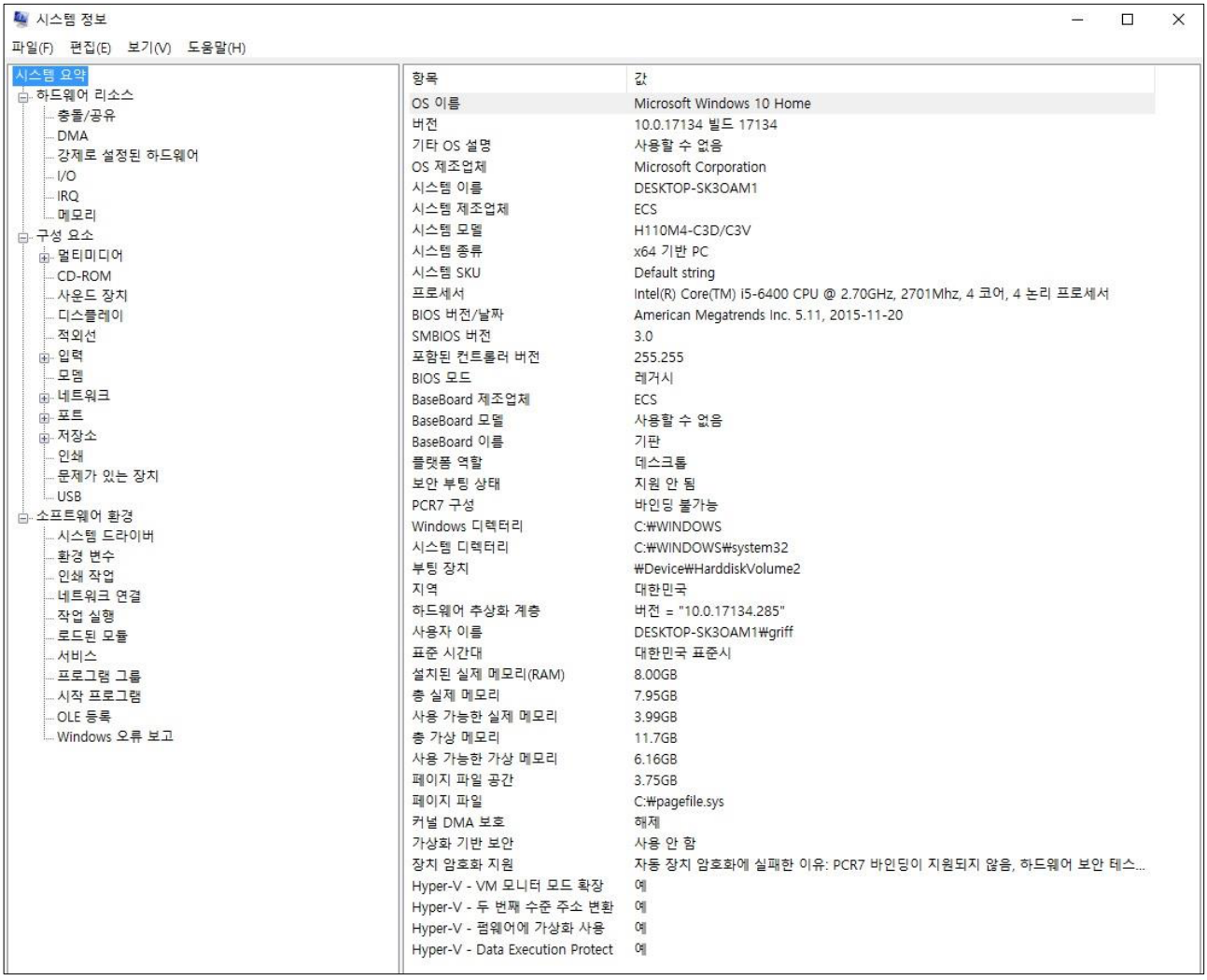
Ⅰ 실험 계획

1. 실험 목적

이번 실험에서는 Greedy Algorithm을 기반으로 하는 Huffman Algorithm을 사용하여 파일을 압축하고 압축을 푸는 프로그램을 제작해 본다.

2. 실험 환경

Huffman Algorithm 코드를 제작하여 임의의 파일을 압축하고 압축을 푸는 실험을 진행할 PC의 시스템과 하드웨어 환경을 조사했다.



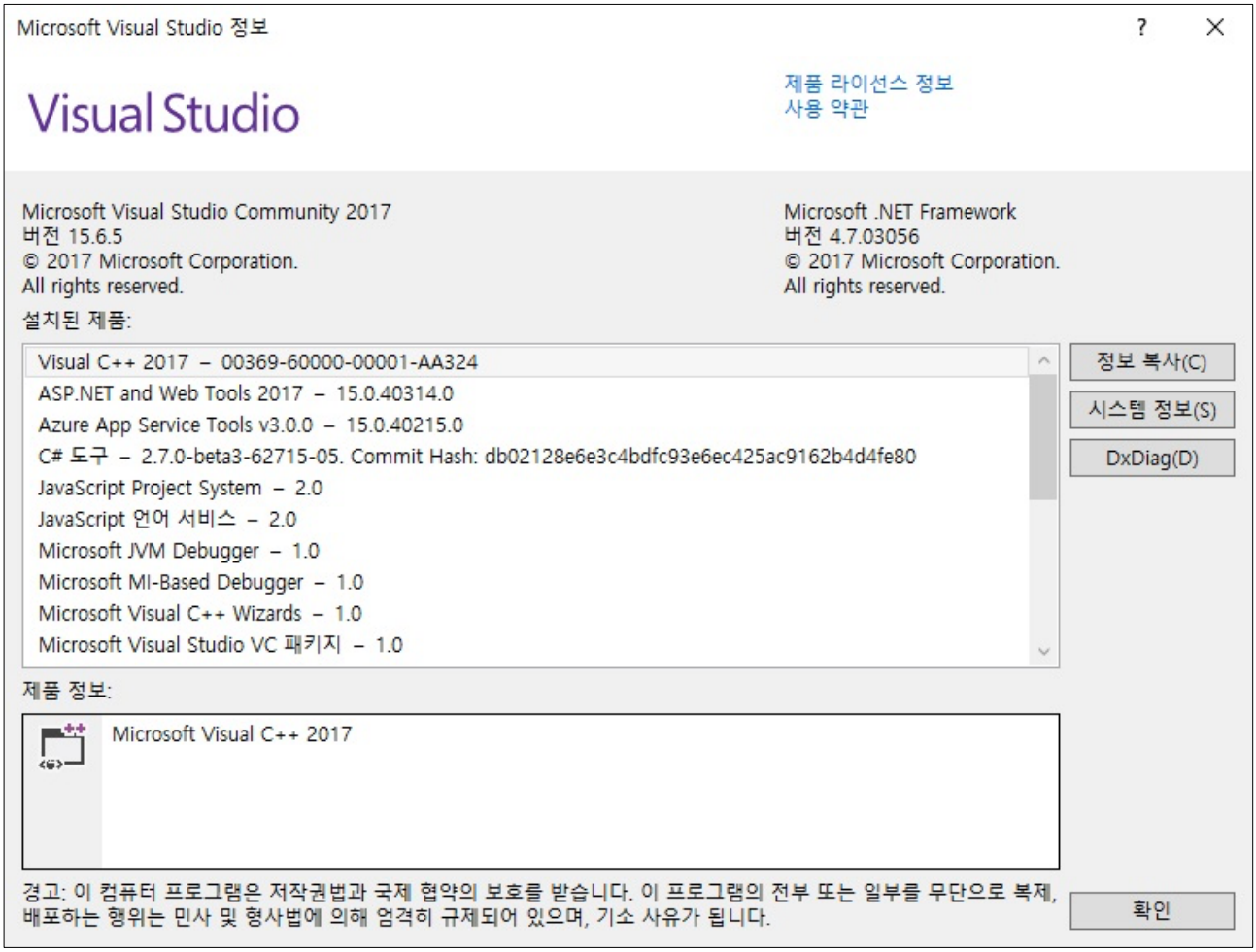
[그림 Ⅰ-1] PC 시스템 환경과 하드웨어 정보

위의 정보를 간략히 요약하자면 *[표 Ⅰ-1]*의 내용과 같다.

항목	값
OS 이름	Microsoft Windows 10 Home
버전	10.0.17134 빌드 17134
시스템 종류	64bit 운영체제, x64 기반 PC
프로세서	Intel(R) Core(TM) i5-6400 CPU @ 2.70GHz, 2701Mhz, 4코어, 4논리 프로세서
설치 메모리	8.00GB

[표 Ⅰ-1] 시스템과 하드웨어 환경 사양 요약

소스 코드를 컴파일(Compile)하여 실행할 프로그램은 Microsoft에서 제작한 Visual Studio 2017이며, 이와 관련한 자세한 정보는 *[그림 Ⅰ-2]*의 내용과 같다.



[그림 Ⅰ-2] Visual Studio 환경 정보

2. 실험 전제

- 소스 코드는 C언어로 작성했으며, 하나의 파일로만 만들었다. Visual Studio 또는 Linux(Unix)에서 기본적으로 제공하는 라이브러리인 stdio.h, stdlib.h, 그리고 string.h를 이용했다.
- 메모리 낭비를 없애기 위해 소스 코드에서 input으로 받는 정보는 malloc 함수를 사용하여 동적으로 메모리를 할당했다.
- Linux 환경에서 파일 명령어와 함께 필요한 두 인자를 사용자가 입력해 main 함수에서 넘겨받을 수 있도록 변수 argc와 argv를 사용했다. Windows에서는 command로 직접 인수를 입력하는 것이 불가능하므로 디버깅 속성 설정에서 전달하는 인자를 입력하여 실험했다.
- 최대로 압축할 수 있는 파일의 용량은 10MB이다.
- 파일 종류별로 압축률이 어떻게 다른지 실험해 보기 위해 각각 pdf, pptx, hwp, txt, psd, jpg 형식의 임의의 6개 파일을 가지고 실험했다.
- 압축 전의 파일과 압축하고 나서 다시 풀었을 때 파일 내용이 같은지를 알아보기 위해 DiffCheckter(https://www.diffchecker.com/diff) 사이트를 이용했다.

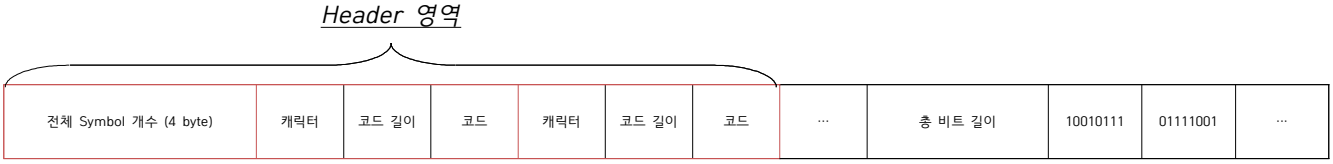
Ⅲ Huffman Algorithm 구현

1. 알고리즘 구현 방법

Huffman Algorithm은 압축하고자 하는 임의의 파일 코드의 문자열에서 출현 빈도가 높은 character의 코드 길이는 짧게, 출현 빈도가 낮은 character(1 byte)의 코드 길이는 길게 만들어서 0과 1의 비트로 이루어진 코드로 압축하는 것이다. 문자열에서 나온 character와 그 출현 수를 조사해서 각각 트리 노드로 만든 후, 출현 수가 낮은 노드 두 개를 선택한다. 그리고 새로운 부모 노드를 생성하여 왼쪽 자식과 오른쪽 자식으로 두 개의 노드를 각각 연결해주고, 부모 노드의 출현 수는 왼쪽 자식과 오른쪽 자식의 출현 수의 합으로 반영한다. 처음에 생성했던 모든 노드가 하나의 트리로 만들어질 때까지 앞의 과정을 반복하며, 완성한 트리(Huffman Tree)에서 왼쪽 자식의 코드에는 '0', 오른쪽 자식에는 '1'을 이어 써서 트리 깊이를 하나씩 내려가며 모든 노드에 관해 0과 1로 이루어진 코드를 완성한다. 압축하고자 했던 문자열을 character마다 읽어서 각 character마다 가진 0과 1로 된 코드로 바꿔서 저장한다.

반대로 압축을 풀 때는 0과 1로 이루어진 압축된 코드를 차례로 읽으면서 앞에서 제작한 트리를 탐색해 간다. 0이 나오면 현재 트리 노드에서 왼쪽 자식으로, 1이 나오면 오른쪽 자식으로 이동하여 leaf 노드(왼쪽 자식과 오른쪽 자식이 모두 없는 노드)가 나올 때까지 트리를 탐색한다. leaf 노드가 나오면 그 노드에 저장된 character를 읽어서 그대로 파일 코드에 쓴다. 이것이 대략적인 Huffman Code의 설명이다.

Huffman Code를 실제로 구현하기 위해서 아래와 같이 작업했다.



1) 압축하기 전에 파일로부터 character를 하나씩 읽어서 Huffman Tree를 제작하기 위해 각 character를 노드로 생성하여 Priority Queue에 넣는다. 노드의 data type은 structure를 사용하여 character인 Symbol, 0과 1로 이루어진 코드인 Bits, 출현 수인 Frequency, structure의 data type을 갖고 Huffman Tree에서 왼쪽과 오른쪽 자식을 가리키는 Left와 Right, 그리고 Priority Queue에 저장하기 위해 사용하는 Link를 structure 변수로 갖는다. QueueFirst는 Priority Queue의 가장 앞에 위치한 노드를 가리키며, TreeRoot와 TempRoot는 Huffman Tree에서의 root 노드를 가리킨다.

2) Priority Queue는 Linked List로 구현했으며, 우선순위는 출현 빈도가 가장 낮은 노드 순으로 높다. 그래서 List의 앞쪽에는 출현 빈도가 낮은 노드를, 뒤쪽에는 높은 노드를 저장하여 앞쪽부터 뒤쪽으로 갈수록 노드의 출현 빈도가 높아지는 순으로 정렬되도록 한다. 이는 파일로부터 character를 읽고 나서 노드를 생성한 후 Priority Queue에 삽입할 때 List의 가장 첫 번째 노드부터 Link를 타고 가면서 앞의 노드보다 출현 빈도가 높고 뒤의 노드보다는 출현 빈도가 낮은 위치에 삽입하는 것으로 구현했다. 만일 파일에서 character를 읽었을 때 이미 Priority Queue에 삽입된 character가 나오면 해당 character를 갖는 노드의 출현 수를 하나 늘려준다. 전체 생성되는 노드의 개수를 n이라고 할 때, 여기서 걸리는 시간 복잡도는 최대 $O(n)$이다.

3) 파일을 모두 읽으면 2)번에서 생성한 Priority Queue에서 가장 우선순위가 높은 노드를 두 개 뽑아서 새로운 노드를 생성하여 부모 노드의 왼쪽 자식과 오른쪽 자식으로 각각 연결하는 작업을 MakeTree 함수에서 진행한다. List에서 가장 앞쪽에 있는 노드가 우선순위가 높으므로 QueueDelete 함수를 통해 가장 앞쪽의 노드를 반환하고 QueueFirst를 Link가 가리키는 노드로 가리키도록 변경한다. 부모 노드를 생성하여 Huffman Tree의 Sub Tree를 제작하면 이의 부모 노드를 다시 Priority Queue에 넣는다. 이를 Priority Queue의 전체 노드 개수인 CharNum보다 1 작은 수만큼 반복한다. 마지막에는 Huffman Tree의 root 노드만 Priority Queue에 남으므로 이를 뽑아서 반환해준다. 여기서 걸리는 시간 복잡도는 $O(n\log n)$이다.

4) GenerateCode 함수를 실행하여 3)번에서 제작한 Huffman Tree에서 각 노드에 그에 맞는 0과 1로 된 코드를 생성하여 저장한다. Post-Order로 트리를 탐색해 가면서 왼쪽 자식과 오른쪽 자식 중 하나라도 있는 노드를 만나면 왼쪽 자식에는 0을, 오른쪽 자식에는 1을 이어 써서 코드를 저장한다. 여기서 걸리는 시간 복잡도는 $O(n)$이다.

5) 이번 Huffman Algorithm을 구현하는 데 있어서 유의해야 할 점은 바로 Huffman Tree에 관한 정보를 압축하는 파일과 같이 저장해야 한다는 것이다. 압축한 파일을 다시 풀 때 Huffman Tree에 관한 정보를 읽어서 다시 재현할 수 있어야 되기 때문이다. 이를 Header 영역이라고 부를 때, 필자는 여기에 전체 character의 개수, 각 character의 Symbol, 코드의 비트 수, 그리고 코드를 저장하도록 GenerateTreeData 함수에서 구현했다. 그래서 압축한 파일을 풀 때 코드 비트가 0이면 왼쪽, 1이면 오른쪽으로 트리를 비트 수만큼 아래로 탐색 또는 확장해 가면서 Huffman Tree를 복원한다. 앞의 Header 영역에 해당하는 데이터를 압축하기 전에 우선 작성한 후, 파일을 다시 처음부터 읽으면서 character에 해당하는 노드를 찾기 위해 Huffman Tree를 In-Order로 탐색한다. 해당 character의 노드를 찾으면 그 노드에 저장된 코드를 비트로 파일에 저장한다. 주의할 점은 C언어에서는 1 바이트씩만 쓸 수 있으므로 1 바이트에 해당하는 한 개의 buffer(필자는 TempBits 포인터 변수로 구현했음)를 만들어서 트리를 탐색할 때마다 비트를 1개씩 저장하고, 비트가 8개로 꽉 차면 파일에 해당 바이트를 쓰는 작업을 진행한다. In-Order로 Huffman Tree를 파일의 모든 character에 관해 탐색해야 하므로 전체 character 수를 m이라고 할 때, 여기서 걸리는 시간 복잡도는 최대 $O(n \times m)$이다. 전체 비트의 개수를 업데이트해야 하므로 모든 비트 코드를 다 작성하면 처음에 작성했던 비트 수를 저장하는 위치로 돌아가서 총 비트의 개수를 반영해준다. 비트 수를 저장하는 이유는 1바이트는 8개의 비트씩 저장하는데, 만약 마지막 부분에서 비트의 개수가 3개만 필요하다면 압축한 파일을 다시 풀 때 나머지 5개의 비트도 character을 복원하는 데 반영되기 때문에 마지막 character에서 오류가 발생할 수 있다.

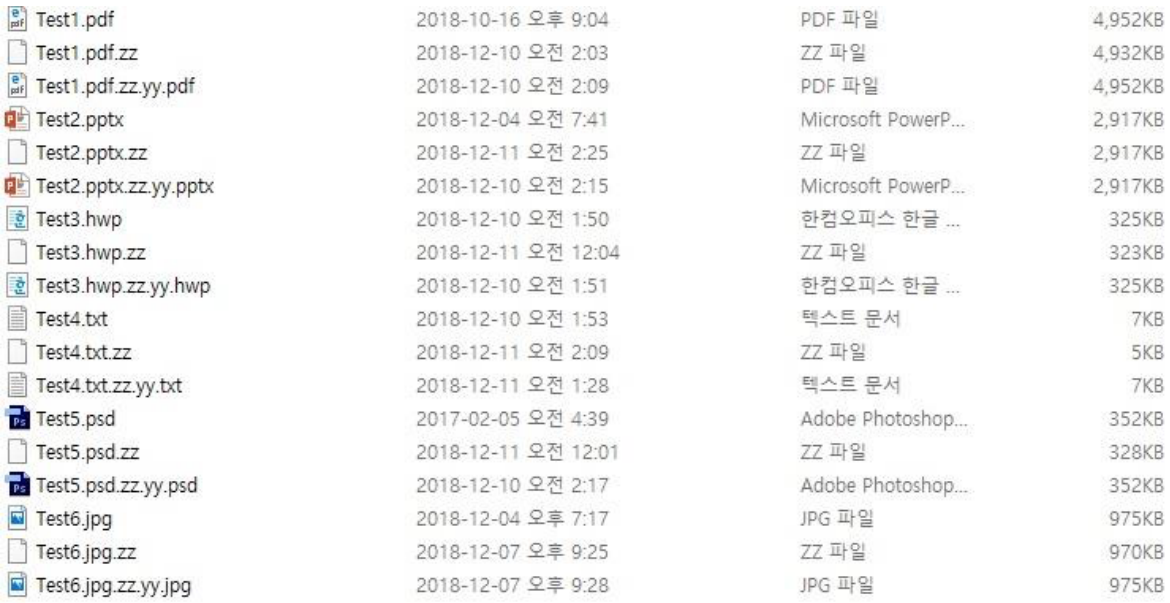
6) 압축한 파일을 다시 복원하기 위해 RecoverTree 함수를 사용하여 Header 영역에 있는 데이터를 읽어서 Huffman Tree를 복원한다. Header 영역 데이터를 읽을 때는 우선 character에 해당하는 1바이트와 코드의 길이에 해당하는 1바이트, 그리고 코드의 길이만큼의 바이트 수를 코드로 읽는다. Huffman Tree를 다 복원한 이후에는 4바이트만큼 전체 비트 수를 읽는데, 이는 앞서 비트 수를 integer data type으로 저장했기 때문이다. 그리고 압축된 파일을 1 비트씩 읽으면서 트리를 탐색해 나가고, leaf 노드까지 도달했으면 해당 노드에 저장된 character를 읽어서 파일에 쓴다.

완성한 Huffman Algorithm 프로그램을 가지고 실제로 파일을 압축할 때 압축하고자 하는 파일의 용량이 커질수록 시간이 굉장히 오래 걸린다. 이는 프로그램에서 걸리는 시간 복잡도를 살펴보면 알 수 있다. 압축하고자 하는 파일의 전체 character(바이트) 수를 m, 실제 Huffman Tree를 제작했을 때의 전체 노드의 개수를 n이라고 할 때, 위의 5)번 과정에서 시간 복잡도가 $O(n \times m)$만큼 걸린다. 이는 읽고자 하는 파일의 바이트 수가 늘어날수록 걸리는 시간이 매우 커질 수 있음을 알려 준다. 만약 여기서 시간을 줄이고자 한다면 Huffman Tree를 탐색하는 데 걸리는 시간 복잡도 $O(n)$을 줄이는 수밖에 없는데, 시간 복잡도를 $O(\log n)$만큼 줄이려면 Balanced Tree 형태를 지닌 Heap로 구성해야 하지만, 이는 Huffman Tree의 구성 원칙에 위배될 수 있기 때문에 실제로 구현해볼 수 없었다.

Ⅱ Performance 실험

1. Performance 실험

파일 유형별로 압축률을 조사하기 위해 pdf, pptx, hwp, txt, psd, jpg 형식의 임의의 파일들을 압축하는 실험을 진행했다. (압축 푼 파일은 확장자를 다시 붙였다.)



파일 종류	파일명	본 파일 용량	압축 파일 용량	압축률
pdf	Test1.pdf	4,952KB	4,932KB	1.01%
pptx	Test2.pptx	2,917KB	2,917KB	1%
hwp	Test3.hwp	325KB	323KB	1.01%
txt	Test4.txt	7KB	5KB	1.4%
psd	Test5.psd	352KB	328KB	1.07%
jpg	Test6.jpg	975KB	970KB	1.01%

[표] 파일 종류에 따른 압축률 비교

압축률은 본 파일 용량에서 압축 파일 용량을 나눈 값이며, 소수 셋째자리에서 반올림했다. 본 파일 용량 크기가 작은 txt 파일에서 가장 높은 압축률을 보였으며, psd가 두 번째로 높은 압축률을 보였다. pdf, hwp, jpg는 모두 압축률이 1.01%로 근접했으며, pptx의 압축률은 1%로써 압축 전과 후의 파일 용량이 모두 같았다. 전반적으로 Performance가 좋지 않음을 알 수 있다. 그리고 상대적으로 파일 용량이 작을 때, 그리고 텍스트로만 또는 이미지로만 이루어진 파일일 때 가장 높은 압축률을 보이는 경향을 유추할 수 있다. 이는 pdf나 pptx, hwp의 경우 이미지와 텍스트가 섞여 있는 경

우가 일반적이며, 실제 실험한 파일도 이 경우에 속했다.

Ⅲ 결론

1. 결론 및 논의

- Huffman Algorithm을 구현하기 위해 Priority Queue와 Tree 자료구조를 사용했다.
- 알고리즘에 의해 압축한 파일을 풀 때 Huffman Tree에 관한 정보가 필요하다. 그래서 이와 관련한 데이터를 파일의 앞 Header 영역에 작성해서 파일을 풀 때 트리를 복원 가능하다.
- 전반적으로 압축률이 1%를 웃도는 수치이며, Performance가 그리 좋지 않다.
- 텍스트로만 이루어지거나 이미지로만 구성된 파일의 압축률이 그렇지 않은 파일보다 비교적 높음을 알 수 있다.
- 파일 용량이 작을수록 더 높은 파일 압축률을 보이는 경향이 있다.

2. 추가 의견

- 이번 프로그램의 Performance를 증진시키기 위해 2바이트로 문자를 읽어서 huffman Tree를 구성하는 것도 시험해보았으나, 막상 2바이트의 문자열이 중복되는 경우가 생각보다 흔치 않아서 압축 효율이 좋지 않고 오히려 압축했을 때 용량이 더 늘어나는 경우도 발생했다. Test4.txt를 2바이트로 읽어서 압축했을 때 용량은 10KB로 기존 용량보다 3KB를 더 차지하게 된다. 그러나 파일 용량이 상대적으로 클 때는 중복되는 문자열이 발생하는 경우가 적어서 1바이트로 읽을 때의 경우와 차이가 거의 없고 오히려 각 문자열의 코드 길이를 줄일 수 있기 때문에 압축률이 상대적으로 더 좋을 것으로 예상한다.