알고리즘

과제 3

소프트웨어학부

2019203021

이승헌

1-1. 압축 해제 방법 – 디코딩

압축 파일인 htb 파일을 이진파일 입출력을 이용해 읽어 버퍼에 쓴 후, 이전에 디자인한 htb 파일 구조에 맞춰 버퍼에 있는 것을 활용해 헤더를 해석한다.

파일의 구조는 이하와 같다.

버퍼의 첫 번째는 마지막 1바이트의 유효 비트 개수, 두 번째는 htb 파일에 있는 문자 종류의 개수, 그리고 세 번째부터 3+3x(버퍼의 두 번째에 있는 값)-1번째 까지는 (문자, 유효 비트 수, 인코딩된 Huffman Code)가 저장되어 있고, 그 이후부터 끝까지는 압축된 텍스트가 저장되어있다.

하지만 소스 코드 파일에는 버퍼의 세 번째 값은 건너뛰고 네 번째 값부터 읽고 있는데,

무슨 이유인지는 불명이나 구현한 압축 알고리즘을 이용하면 버퍼의 세 번째 값에는

항상 3이 저장되어 있어 헤더를 제대로 해석하지 못하였다. 그래서 네 번째 값부터 읽는 것이다.

Huffman Tree를 압축 해제에서는 직접 구성하지는 않았으나, 이전에 압축 과정에서

완성한 Huffman Tree의 결과를 가져온 방법은 이하와 같다.

문자의 유효 비트 수를 읽어서, 인코딩된 Huffman Code에서 유효한 비트를

Left Shift와 AND 연산을 이용해 계산하였다.

구체적인 예를 들면

1

A2(0100000)B1(10000000)C2(00000000)

0001110110000000

이렇게 저장된 htb 파일이 있다고 하자. 그리고 A를 읽을 때는 A를 symbol에,

2를 significant에, char 형태로 되어 있는 0100000을 byte\_code에 저장하며, 또

A의 유효 비트를 나타내는 문자열을 code라고 하자.

그러면 반복문을 이용해서7비트만큼 Right Shift한 byte\_code를 1과 AND 연산하여 가장 왼쪽 비트를 얻어내고, 문자열에 대한 +연산을 이용해 code에 +연산을 하고, 다음 하위 비트에 대해 같은 연산을 Right shift 횟수만 감소시켜 나가며 수행하여 significant 횟수만큼 반복한다.

그 결과로 code에는 “01”이라는 문자열이 저장될 것이다. 이 때 C++의 map을 이용한다.

이 map을 decoded\_code라고 한다면 decode\_code[code]=symbol 이렇게 저장한다.

유효 비트를 표현한 문자열에 대해 한 문자가 대응되도록 하는 것이다.

위 과정을 B와 C에서도 수행해 map 자료구조에 모든 Huffman Code가 모든 문자에 대응되도록 만든 것이다.

구현한 압축해제 프로그램은 디코딩을 위 예시와 같이 진행한다.

1-2. 압축 해제 방법-실제 압축해제

압축 해제 과정은 아래 예제를 프로그램의 구현 방법대로 압축 해제 하는 과정으로 설명하겠다.

1

A2(0100000)B1(10000000)C2(00000000)

0001110110000000

decoded\_code 자료구조는 다음과 같이 문자열을 문자에 대응시킨다.

decoded\_code[“01”]=’A’

decoded\_code[“1”]=’B’

decoded\_code[“00”]=’C’

1. 버퍼에 저장된 한 바이트를 읽어서 target에 저장한다. 여기서 target은 00011101이다.

2. target은 파일의 마지막 부분을 읽어온 것이 아니므로, 이하 과정으로 진행된다.

target을 7비트 Right Shift한 값을 1과 AND 연산하여 가장 왼쪽 비트를 bit에 저장하고,

그 bit를 전역 문자열 tmp\_code에 +연산으로 덧붙인다.

위 과정을 가장 오른쪽 비트를 읽을 때까지 반복한다.

이 때, 위 과정을 반복하면서 계속해서 decoded\_code[tmp\_code]를 했을 때 대응되는 문자가 있는지 체크한다. 만약 대응되는 문자가 있다면, hfd 파일에 대응된 문자를 write한 다음 tmp\_code를 비운 뒤 반복을 계속해서 수행한다.

이 단계에서는 00, 01, 1, 1, 1, 01로 끊어서 읽을 수 있으므로 CABBA를 hfd에 입력하였다.

또한 끝까지 모두 끊어서 읽었으므로 이 시점에서 tmp\_code는 빈 문자열이다.

3. 1, 2를 파일의 끝부분을 읽을 때까지 반복한다.

4. target이 파일의 마지막에 있는 1바이트를 읽은 경우, 버퍼의 첫 번째 값인 마지막 1바이트의 유효 비트 수를 이용하는 과정이 추가된다.

target의 8비트부터 8-(마지막 1바이트이 유효 비트수)비트까지 추출하고 tmp\_code에 덧붙이는 과정을 진행한다. 즉, 2의 과정에서 바뀌는 것은 8비트 전부를 추출하지 않도록 내부 반복문의 반복 횟수를 조절하는 것 뿐이다. 만약 추출하고 덧붙이는 과정에서 tmp\_code에 대응되는 문자가 있다면 hfd 파일에 그 문자를 쓰고 그 즉시 내부 반복문을 종료한다.

위 예시로 보면 target은 1000000인데, 버퍼의 처음 값이 1이고 tmp\_code는 비어있으므로

tmp\_code=”1”이고, 이는 B에 대응된다. 따라서 hfd에 d를 쓰고 그 즉시 내부 반복을 종료한다.

그 결과, hfd에는 CABBBAB라는 값이 적히게 된다.

2. 실제 압축 해제 결과 텍스트, 스크린샷, 소프트웨어, 웹 페이지이(가) 표시된 사진

자동 생성된 설명

winMerge프로그램을 이용해 차이점을 보았다. (다른 파일은 크기가 너무 커 생략하였다.)

파일 크기로 살펴본 결과 압축 해제를 했을 때 크기가 1% 내외로 더 커지는 것을 관찰했고,

파일이 커지면 커질수록 차이점이 많아지는 걸 관찰 할 수 있었다.

완전히 다르게 압축 해제되지는 않고 유사한 점이 어느 정도는 있지만,

압축을 해제하는 것 뿐인데 파일이 변해버렸다는 점에서 완벽하게 실패했다고 볼 수 있다.

target의 bit를 하나씩 하나씩 출력한 텍스트 파일과 hft 파일을 대조해본 결과,

압축 해제 프로그램에는 문제가 없었다는 점을 알 수 있었다.

정상적으로 압축되었는데 엉뚱한 문자로 잘못 판단한 것이 아니라 파일에 저장된 그대로 해석하고 있음을 발견하였기 때문이다.

다시 말해, 압축하는 과정에서 무언가 문제점이 발생한 것이다.

디코딩 부분에서 언급했듯 압축 프로그램의 코드에는 무의미하게 데이터를 write하는 코드는 없었음에도 불구하고 항상 3번째 바이트에는 3이라는 숫자가 저장되어있었다.

이러한 점을 미루어 보아 압축 프로그램이 실제 문자열을 압축하는 과정에서 결함이 있는 코드를 가지고 있을 확률이 높다고 생각이 된다.

모든 부분이 엉뚱한 문자로 압축 해제되어 있지는 않았고, 또한 어떤 위치를 기점으로 엉뚱한 문자로 압축 해제를 했더라도 어느 순간 또 다시 정상적으로 압축 해제를 하는 모습을 보이기 때문에 압축 프로그램에서 원인을 파악할 수 없었다.