Huffman Coding for File Compression

과목 : 알고리즘 설계 및 분석

분반 : 2 반

교수님 : 소정민 교수님

학번 : 20141284

이름 : 이기현

1. 프로그램 실행 환경

* OS : Linux version 4.4.0-164-generic (gcc version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.10))
* CPU Model : Intel® Xeon® CPU E5-2630 v4 @ 2.20GHz
* CPU MHz : 1632.1215
* Cache size : 25600 KB
* RAM : 65862640KB (= 65GB)

1. 압축 파일 설명

- 압축 파일은 3부분으로 구성된다.

- binary string의 size => 압축 string의 각 char의 하위 7비트를 모두 모아 만든 binary string의 사이즈

- 0 ~ 127 character의 빈도 횟수 => decompression 때 Huffman tree를 생성하기 위함

- 압축 string => 실제 압축 string

1. 구조체 및 함수 설명

struct tree\_node

int cnt; // 빈도수를 위한 변수

char x; // tree의 루트에 있을 때 해당 문자

tree\_node\* left, \* right // Huffman tree에서 자손을 향하는 링크

* Huffman tree 의 각 노드를 나타내는 노드이며 (빈도수, 문자) 의 pair나 특정 노드의 포인터를 이용한 생성자를 추가로 구현해 놓았다.

struct cmp

bool operator() (tree\_node a, tree\_node b) { return a.cnt > b.cnt; }

* Huffman tree 를 만들기 위해 빈도수를 기준으로 minimum heap을 만들기 위한 구조체로 STL priority queue의 compare function으로 들어간다.

void make\_huffman\_tree (vector<pair<int, int> > &cnt, tree\_node \*\* Tree)

* Input : cnt – 128 character의 빈도수 배열, Tree – Huffman tree 의 루트 노드
* 빈도수 배열인 cnt와 빈 트리인 Tree를 인자로 받아 Huffman tree를 생성해준다.
* 빈도수를 기준으로 하는 minimum heap에 모든 문자의 빈도수를 넣은 다음, 빈도수가 작은 문자 2개를 꺼내어 합쳐 각각을 자식으로 하는 이진 트리를 생성한다. 그 후, 두 문자의 빈도수를 합쳐 해당 이진 트리 루트 노드의 빈도수로 설정하고 다시 heap에 넣어준다.
* priority queue 내부에 더 이상 합칠 노드가 없을 때까지 실행한다.
* 최종적으로 인자로 받은 Tree에 여태까지 합친 이진 트리를 덮어 씌워준다.
* 만약 priority queue에 최초로 들어간 노드가 1개라면 Huffman tree의 루트 노드에 문자가 쓰이기 때문에 나중에 binary string을 만들 때 문제가 생겨 예외처리를 해준다.

void make\_dic (map<char, string> &dic, tree\_node \*tree, string & s)

* Input : dic – 문자별 binary string을 저장하기 위한 map, tree – Huffman tree, s – binary string을 위한 temporary string
* Huffman tree를 DFS식으로 탐색하며 leaf노드에 도착하면 해당 leaf 노드의 문자의 binary string을 dic에 저장해준다.
* 재귀 형태로 구현하였으며 자식 노드로 재귀호출 전에 s에 0 또는 1을 이어붙인 다음 재귀호출을 하고, 재귀 호출이 끝나면 다시 제거해준다.

void compression ()

* input file 로부터 내용을 읽어 Input string을 저장하고 그 과정에서 각 문자의 빈도수 역시 계산해준다.
* 계산한 빈도수를 이용해 Huffman tree를 만들고 각 문자의 binary string을 저장한다.
* Input string을 처음부터 읽으면서 각 문자를 binary string으로 변환한다.
* 변환된 binary string의 사이즈와 0~128 문자의 각 빈도수를 출력해준다.
* binary string 으로 변환된 Output file을 7개 단위로 잘라서 하나의 char 변수에 넣고 가장 앞의 bit는 1로 설정해주어 출력한다. 8개씩 자르지 않고 가장 앞의 bit을 1로 설정한 이유는 char 변수에 bit를 채웠을 때 EOF(26)이 될 경우 압축/해제에 문제가 생기기 때문이다.

void decompression ()

* 변환 file 로부터 binary string의 사이즈와 0 ~ 128 문자의 각 빈도수를 입력 받는다.
* 받은 빈도수로 Huffman tree를 우선적으로 생성해준다.
* 변환된 string을 입력 받아서 임시 string에 보관했다가 한 char 씩 읽으며 하위 7bit를 각각 0 또는 1로 표현한 binary string을 받는다.
* binary string을 앞에서부터 읽으며 Huffman tree를 루트부터 내려가고 만약 leaf 노드에 도착하면 해당 leaf 노드의 문자를 출력, 다시 루트로 돌아가 진행하는 반복을 한다.

4. 0~128 ascii random

- Input data 설명

* size : 10000byte(10kb), 100000byte(100kb), 1000000byte(1mb), 10000000byte(10mb)
* data : ascii 26을 제외한 random character
* 각 사이즈별 다른 데이터로 3번 시행
* 수행 결과 및 압축률 비교 (pass 여부 / 압축률)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 10kb | 100kb | 1mb | 10mb |
| 1차 | 0.974 | 1.001 | 1.004 | 1.005 |
| 2차 | 0.987 | 1.001 | 1.005 | 1.005 |
| 3차 | 0.977 | 1.002 | 1.005 | 1.006 |

- Comment

* seed를 임의로 받아서 랜덤 데이터를 만들어 사용하였다. 데이터를 랜덤으로 만들고 128의 modulo를 이용하여 값을 구했다. 랜덤 데이터를 만들었는데 압축률이 1에 가까이 즉, 거의 되지 않았다.
* 이유를 찾기 위해 .zz 파일을 열어 각 문자의 빈도수를 확인하였다. 분명 임의의 seed에 rand함수를 이용하였는데 대부분 문자의 등장 빈도수가 거의 비슷비슷 했다. 그렇기 때문에 Huffman tree를 만들었을 때, 모든 문자 즉, leaf가 마지막 레벨에 몰려 있는 full binary tree 형태로 생성되었을 것이며, 그렇기 때문에 128개의 ascii 문자 모두 대략 7bit의 2진 string으로 변환이 될 것이다. 거기에 EOF 방지를 위해 한 char에 7bit씩 나누어 압축을 하였는데 이 때문에 압축 string의 길이가 입력 string의 길이와 차이가 거의 나지 않아 위와 같은 압축률이 등장한 것으로 추정된다.
* 또한 한 char에 7bit를 넣어 압축하였기 때문에 실제 압축된 binary string을 이용하여 압축률을 예상해보자면 8/7 = 1.142857을 곱하여 대략 1.143 ~ 1.149 정도로 예상된다.

5. alphanumeric ascii random

- Input data 설명

* size : 10000byte(10kb), 100000byte(100kb), 1000000byte(1mb), 10000000byte(10mb)
* data : 알파벳 대문자, 알파벳 소문자, 숫자
* 각 사이즈별 다른 데이터로 3번 시행
* 수행 결과 및 압축률 비교 (pass 여부 / 압축률)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 10kb | 100kb | 1mb | 10mb |
| 1차 | O / 1.112 | O / 1.167 | O / 1.172 | O / 1.174 |
| 2차 | O / 1.123 | O / 1.170 | O / 1.174 | O / 1.173 |
| 3차 | O / 1.124 | O / 1.166 | O / 1.173 | O / 1.174 |

- Comment

* 128개의 모든 ascii 문자를 이용하였을 때는 빈도가 비슷하게 나와 데이터양을 줄여보았다.
* alphanumeric 문자인 알파벳 대/소문자와 숫자를 이용하여 랜덤 데이터를 뽑았다.
* 데이터 결과를 확인하기 전에 빈도수를 먼저 확인하였는데 이번에도 역시나 빈도수가 비슷하였다. 그래서 압축률을 확인하였는데 이전 테스트보다는 괜찮아졌다.
* 빈도수가 비슷할 때, full binary tree형태로 Huffman tree가 생기는 것은 비슷했지만 사용한 문자가 62개로 절반정도로 줄었기 때문에 각 문자의 변환 binary string은 약 6bit정도 된다. 그렇기 때문에 이전보다 압축string을 6/7정도로 줄일 수 있었다. 즉, 압축률을 7/6 = 1.166배정도 늘릴 수 있었다. 계산해보면 1.167 ~ 1.170 정도로 대략 알맞은 압축률 결과가 나온 것을 확인할 수 있었다.
* 이 또한 한 char의 한 bit은 의미 없는 1이 들어갔기 때문에 실제 bit의 압축률을 따져보면 8/7 = 1.142857 을 곱해서 1.334 ~ 1.338 정도의 압축률을 예상할 수 있다.

6. one-character-ascii random

- Input data 설명

* size : 10000byte(10kb), 100000byte(100kb), 1000000byte(1mb), 10000000byte(10mb)
* data : 0~128 ascii 중 26을 제외한 임의의 character
* 각 사이즈별 1번 시행
* 수행 결과 및 압축률 비교 (pass 여부 / 압축률)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 10kb | 100kb | 1mb | 10mb |
| 1차 | O / 5.896 | O / 6.870 | O / 6.987 | O / 6.998 |

- Comment

* 최고의 압축률을 확인하기 위해 한 문자로만 구성된 Input을 만들어 테스트 해보았다.
* 문자가 1개이기 때문에 압축을 하면 해당 문자가 1bit 짜리 binary string으로 변환된다. 그렇기 때문에 총 7개의 문자가 1byte로 압축되는 효과를 얻을 수 있다. 하지만 빈도수를 압축 파일에 기록해놓기 때문에 작은 size의 파일에서는 7배의 압축률을 얻을 수 없었다. 10mb 정도로 파일 사이즈가 커졌을 때는 압축률이 7에 수렴하는 것을 확인 할 수 있었다.
* 이 입력 역시 binary string으로 bit의 압축률을 예상해보면 8/7을 곱해서 대략 8배로 예상할 수 있다.

7. article data

- Input data 설명

* size : 44726byte(45kb), 447260byte(450kb), 4472600byte(4.5mb)
* data : 알파벳 대/소문자 및 숫자 및 기타 기호
* 1번 시행
* 수행 결과 및 압축률 비교 (pass 여부 / 압축률)

|  |  |  |  |
| --- | --- | --- | --- |
|  | 45kb | 450kb | 4.5mb |
| 1차 | O / 1.521 | O / 1.538 | O / 1.540 |

- Comment

* 랜덤으로 만든 데이터는 빈도수가 대부분 비슷하게 나와서 실제 압축률을 기대해보기가 힘들었다.
* 영문으로 된 기사를 3개 정도 합쳤으며, 0 ~ 127의 ascii 문자가 아닌 문자가 많아서 일일이 수정하기 너무 힘들어 더 합치기는 어려웠다. 그리하여 450kb 나 4.5mb 사이즈의 데이터는 45kb의 데이터를 반복하였다.
* 확실히 사이즈가 커질수록 압축률이 증가함을 볼 수 있었고 기존에 랜덤데이터로 만들어 빈도수가 비슷했을 때보다 압축률이 증가했음을 확인할 수 있었다.