**CSE3081-2: Design and Analysis of Algorithms (Fall 2019)**

**Machine Problem 3: Huffman Coding for File Compression**

학과 : 컴퓨터공학 학번 : 20181668 이름 : 이예진

**1. Experiment environment**

\*The hardware specification of the machine

-CPU speed : 1.80GHz (8CPUs), ~2.0GHz

-Memory size : 8192MB RAM (8GB)

-OS type and version : Windows 10 Home 64비트 (10.0, 빌드 18362)

-개인 노트북(LG전자 올뉴그램 15Z980-GA70K, 코어i7-8550U)으로 모든 실험을 진행했다.

-Linux 서버에서 개인 계정으로 모든 실험을 진행했다.

**2. Description of algorithm**

(1) 사용한 자료 구조

-TreeNode 형 구조체

변수로 빈도수를 저장하는 weight, 데이터를 저장하는 alpha, 허프만 코드를 저장하는 code 배열, 허프만 코드의 길이를 저장하는 depth, 노드의 왼쪽 자식 노드와 오른쪽 자식 노드를 가리키는 포인터 left\_child와 right\_child가 있다.

-Node 형 구조체

변수로 빈도수를 저장하는 f, 데이터를 저장하는 data, 허프만 코드를 저장하는 code 배열, 허프만 코드의 길이를 저장하는 size가 있다.

-element 형 구조체

Treenode형 ptree와 int형 key를 변수로 가지는 구조체이다. 힙의 원소로 사용된다.

-HeapType 형 구조체

element를 원소로 갖는 heap 배열과 heap 배열의 원소 개수를 저장하는 변수 heap\_size를 가지고 있다. heap 형태의 Priority Queue를 구현하기 위해 사용되는 자료형이다.

-최소 힙과 트리

문자가 입력 파일에 출현한 빈도수를 기준으로 최소 힙 성질을 갖는 트리를 생성하여 이용했다. 각 문자에 이진 코드를 부여했는데 왼쪽 자식에 ‘0’, 오른쪽 자식에 ‘1’을 사용했다. 상향식으로 만드는 이진 트리로 greedy 방법을 이용한 전이진트리(리프노드를 제외한 나머지 모든 노드들은 자식을 2개씩 갖는다.)이다.

트리를 만드는 방법 : 각 문자가 개별적인 트리인 상태에서 빈도수가 작은 두 개의 트리를 합치는 과정을 반복한다. 합쳐지는 두 개의 트리는 자식 노드를 갖는 부모 노드를 생성하는데, 이때, 부모 노드의 빈도수는 두 자식 노드 빈도수의 합이다.

(2) 압축 파일의 포맷

-compressed 된 파일

[입력 파일에서 사용된 문자의 개수(int형)]/[문자][문자에 대한 허프만 코드의 길이][허프만 코드](모두 char형)……[문자][문자에 대한 허프만 코드의 길이][허프만 코드][\n(줄바꿈)]

//여기까지 허프만 트리에 대한 정보를 저장한다.

[비트의 총 개수(4 바이트)][문자열을 8비트씩 끊어 저장(비트 단위)]

//여기까지 입력 파일의 내용을 비트로 변환하여 저장한다.

-decompress 과정

decompress 함수를 이용해서 .zz로 끝나는 입력 파일의 압축을 풀어 원래의 파일로 되돌리고 .yy로 끝나는 파일을 생성한다. 입력 파일을 열어 사용된 문자의 개수를 Symbol 변수에 저장한다. %\*c을 이용해서 변수 뒤에 있는 ’/’ 문자를 무시한다. ChAndLen 배열을 선언하고 char형으로 적혀있는 문자와 문자에 할당된 허프만 코드의 길이를 읽는다. buf 배열을 선언하여 허프만 코드의 길이만큼 허프만 코드를 읽는다. buf의 마지막에 ‘\0’을 추가한다. current 포인터 변수로 트리를 따라가면서 ‘0’이면 왼쪽으로, ‘1’이면 오른쪽으로 이동하고 마지막에 ChAndLen[0]에 저장되어 있는 문자를 대입해준다. 이 과정을 반복하여 Symbol 개의 문자로 허프만 트리를 복구하여 생성한다. %\*c를 이용해서 줄바꿈 문자를 무시한다. numBits 변수에 비트의 전체 개수를 읽는다. 이 정보는 4바이트로 저장되어 있기 때문에 int 변수에 저장한다. buf2와 이 버퍼의 인덱스인 bidx를 선언한다. numBits만큼 문자들을 읽어서 비트로 적혀 있는 정보를 다시 char 문자들로 복구하는 과정을 거친다. 비트는 8비트 단위로 되어 있기 때문에 이중 for문의 안쪽 for문은 8번 반복한다. buf2에 저장된 정보와 0x80을 and 연산하고 시프트 연산을 하여 허프만 트리의 알맞은 자리를 찾는다. 그 다음 current에 저장된 문자를 출력 파일에 출력해주고 current를 다시 허프만 트리의 루트에 위치시켜준다.

(3) implementation issues or design choices

-파일에서 NULL 문자(아스키 코드 0번)도 효율적으로 읽기 위해 fgets나 fread의 사용을 자제하고 fscanf을 주로 사용했다.

-strlen 함수는 ‘\0’ 전까지의 길이만을 셀 수 있기 때문에 문자열 중간에 널 문자가 포함되어 있으면 함수를 사용하지 않고 직접 문자열의 길이를 세었다.

-open\_input 함수에서 입력 파일이 존재하는 경우 반환 값을 2가 아닌 1로 설정하여 NULL 문자만 있는 파일을 읽을 때 오류가 발생하지 않도록 했다.

(4) 변수

-Node \*arr : 입력 파일에서 사용된 문자들의 정보를 저장하는 배열이다.

(5) 함수

-init, insert\_min\_heap, delete\_min\_heap : init는 자료형이 HeapType인 배열을 초기화하는 함수이다.각각 최소 힙 성질을 유지하면서 힙에 노드를 추가하고 삭제하는 함수이다.

-make\_tree, destroy\_tree : 트리를 만들고 삭제하는 함수이다.

-huffman\_code, sub\_order, level\_order, huffman\_tree : 허프만 트리를 이용해서 각 문자에 허프만 코드를 부여할 때 사용되는 함수들이다. huffman\_tree에서 1번 이상 나온 문자들로 최소 힙 성질을 유지한 트리를 만든다. huffman\_code 함수를 호출하여 허프만 코드를 생성하고 트리의 최대 깊이를 저장하는 변수 maxdepth에 최대 깊이를 저장한다. level\_order와 sub\_order 함수를 이용해서 허프만 트리를 모두 순회하고 각 노드의 값들을 arr에 대입한다.

-compress, decompress : 각각 압축을 하고 압축을 푸는 기능을 하는 함수이다. compress 함수는 입력 파일에서 사용된 문자의 개수를 저장한 변수 charnum을 출력파일에 쓰고 그 개수만큼 arr에 저장되어 있는 정보를 쓴다. 그 문자와 허프만 코드의 길이, 허프만 코드를 모두 char형으로 쓴다. 줄을 바꾸어 location 변수에 현재 위치를 저장한다. bitbuf 배열에 한 줄씩 문자열을 읽는다. allflg 변수는 파일 끝까지 읽었을 때 1으로 설정되고 nullflg 변수는 읽은 문자열에 널 문자가 있는 경우 1으로 설정된다. 반복문을 이용해서 읽은 글자를 arr배열에서 찾고 허프만 코드를 huffmanCode 배열에 저장한다. 비트 연산을 수행하여 bitbuf에 비트 단위로 저장한 다음 출력파일에 쓴다. 8비트 단위로 써주는데 flag가 1인 경우에는 8비트가 되지 않았지만 남은 비트들도 출력 파일에 쓴다. 마지막으로 location에 저장했던 위치에 전체 비트수인 totalbit를 쓴다.

decompress 함수는 (2)에서 설명했다.

-open\_input : .txt 형태의 입력 파일을 열어서 문자의 빈도수를 계산하고 charnum에 중복을 제외하여 사용된 문자의 개수를 저장하고, number에 중복을 포함하여 파일에 존재하는 문자의 총 개수를 저장한다. arr를 초기화하여 huffman\_tree 함수를 호출하고 입력 값에 알맞은 허프만 코드를 생성한다.

**3. 실험**

(1) 실험 목적

허프만 코딩을 기반으로 파일 압축 유틸리티 프로그램을 작성한다. 파일을 압축하고 원래의 파일로 되돌리는 유틸리티 프로그램이다. 압축을 위해 그리디 알고리즘인 허프만 코딩을 이용해야 한다.

(2) 실험 결과

\*compression ratio = original file size / compressed file size

EX1> 적은 문자가 사용된 파일

-small1.txt

[1] original file size

19.6KB

[2] compressed file size

2.59KB

[3] compression ratio (KB 단위)

19.6 / 2.59 = 7.56756757 (약 7.57)

-small2.txt

[1] original file size

1.22MB

[2] compressed file size

165KB

[3] compression ratio (KB 단위)

1220 / 165 = 7.39393939 (약 7.39)

-small3.txt

[1] original file size

9.76MB

[2] compressed file size

1.28MB

[3] compression ratio (KB 단위)

9760 / 1280 = 7.625 (약 7.63)

EX2> 많은 문자가 사용된 파일

-large1.txt

[1] original file size

19.2KB

[2] compressed file size

13.6KB

[3] compression ratio (KB 단위)

19.2 / 13.6 = 1.41176471 (약 1.41)

-large2.txt

[1] original file size

1.07MB

[2] compressed file size

706KB

[3] compression ratio (KB 단위)

1070 / 706 = 1.51558074 (약 1.52)

-large3.txt

[1] original file size

10.1MB

[2] compressed file size

6.21MB

[3] compression ratio (KB 단위)

10100 / 6210 = 1.62640902 (약 1.63)

\*표 : 단위는 KB이다.



-결론 : 실험 결과를 바탕으로 압축 비율은 일정하지 않은 것을 알 수 있다. small 파일들의 경우, 문자를 2 ~ 3개 정도로 적게 사용했고, large 파일들의 경우, 문자를 ASCII 코드 전체로 사용했다. 결과로 보아 small의 경우가 large의 경우보다 평균적으로 압축 비율이 좋은 것으로 보아 압축이 효율적임을 알 수 있었다.