고급계산이론 HW2 Report

공과대학 컴퓨터공학부 석사과정
2017-20870 오평석

0. 컴파일 방법

c++11 플래그를 주고 컴파일하면 됩니다. Makefile이 같이 들어있어서 make를 입력하여도 됩니다.

1. LZ78 encoding 구현

(1) dictionary 구성

먼저 입력받은 text로부터 dictionary를 구성하여야 합니다. 이는 트라이를 이용하여 효율적으로 구현할 수 있습니다. Text를 한 글자씩 읽으면서 phrase에 넣다가, 이 phrase가 트라이에서 검색이 되지 않을 때 트라이에 insert를 한 후 dictionary에 추가하는 식으로 구현하였습니다. 트라이 구현은 과제1에서의 구현을 그대로 가져왔으며, 각 노드마다 128개의 배열을 가지도록 하였습니다. 이 때 character->integer의 mapping을 담당하는 배열이 필요한데, 편의상 C++ STL의 map을 사용하였습니다.

(2) bitstream 생성

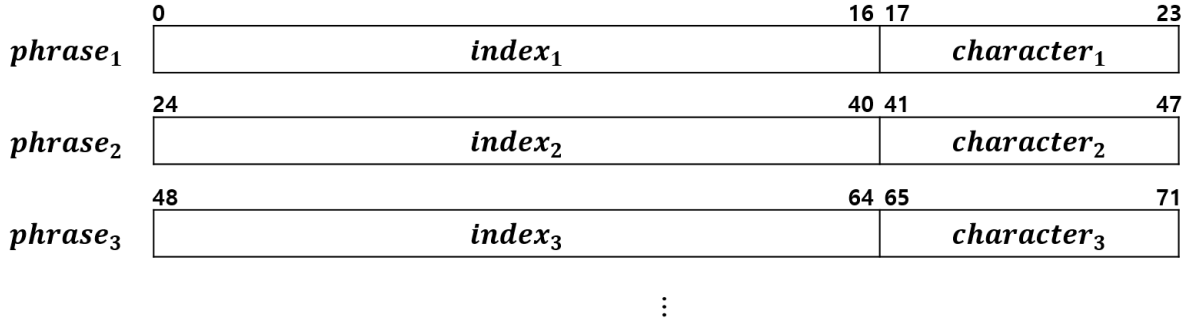
그 다음은 dictionary로부터 bitstream을 생성하여야 합니다. Dictionary는 (index, character)의 배열이므로, 각각을 encoding하는데 사용할 bit를 정한 다음 이걸 그대로 파일에 write를 하면 됩니다. 가장 naïve하게 정하면 index는 4byte, character는 1byte를 할당하여 bitstream의 크기는 5byte * (dictionary의 크기) 가 됩니다.

(3) optimization

압축률을 높이기 위해 다음 두 가지 optimization을 적용하였습니다.

먼저 character에 할당하는 bit를 7-bit로 줄였습니다. 문제에서 정의된 alphabet의 ascii code값은 모두 127을 넘지 않습니다. 따라서 7-bit로도 모든 alphabet을 표현할 수 있습니다.

그 다음 트라이에 들어가는 원소 개수를 제한하였습니다. 트라이가 커지면 dictionary에 들어가는 index의 범위도 같이 커져, index에 더 많은 bit를 할당하여야 합니다. 일반적으로는 Huffman encoding같은 기법을 추가로 적용하는 식의 2-pass encoding으로 해결할 수 있어 보이지만, 이번 과제에서는 다른 encoding의 사용이 금지되어 있습니다. 따라서 index에 할당하는 bit를 줄이기 위하여 트라이의 크기를 제한하였습니다. 이 크기를 얼마나 제한하는지에 따라 압축률이 달라질 것인데, 구현에서는 2^17 = 131,072로 정하여 index를 17-bit로 표현할 수 있도록 하였습니다. 이렇게 index에 17-bit, character에 7-bit를 할당하여 총 24-bit로 dictionary의 각 element들을 encoding 하였습니다.



⋮

2. LZ78 decoding 구현

Decoder 구현은 매우 간단합니다. 앞에서 dictionary의 각 element가 24-bit로 encoding되어있으므로, 3byte를 읽은 다음 이걸 잘 쪼개서 (index, character)로 파싱하면 됩니다. Text의 복원은, 트라이에서 root 노드를 만날 때까지 부모 노드를 타고 올라가면서 character를 phrase에 담은 다음, 이 phrase를 거꾸로 뒤집어 출력하면 됩니다.

3. Example running 결과

조교님이 주신 1개의 example과, 3개의 own example을 준비하여 encoder와 decoder의 동작을 확인하였습니다. sample/ps1_in.txt는 wikipedia에서 여러 항목으로부터 일부분을 발췌하여 엮은 후, 범위에 맞지 않는 문자들을 삭제한 텍스트입니다. sample/ps2_in.txt는 Lipsum generator에서 100,000 bytes를 생성한 후, 이를 20번 반복하여 얻은 텍스트입니다. sample/ps3_in.txt는 input에 중복이 매우 심하게 일어나는 극단적인 텍스트입니다. 이 텍스트들을 input으로 넣어서 encoder와 decoder의 수행시간, output 크기 등을 요약하면 다음과 같습니다.

Input		Encoding			Decoding	Diff
File	Size (B)	Time (s)	Size (B)	Ratio (%)	Time (s)	
ta_in	1555051	0.284	796149	48.8	0.127	Equal
ps1_in	1985492	0.290	931830	53.1	0.108	Equal
ps2_in	2006559	0.211	635400	68.3	0.092	Equal
ps3_in	2096128	0.104	8817	99.6	0.034	Equal

작성한 LZ78 encoder는 일반적인 경우 약 40~50% 근처의 압축률을 보였습니다. Text에 중복이 많이 섞일수록 압축률은 올라갑니다. 만약 input이 극단적으로 중복이 많이 일어나면, 99% 이상을 보여주기도 합니다.

Encoding과 decoding에 걸리는 시간은 개선의 여지가 있지만 오래 걸리지는 않았으며, 모두 2MB 정도의 input에 대해서는 1초 안으로 실행되는 것을 확인할 수 있었습니다.

4. 압축 프로그램과의 비교

압축 프로그램으로는 gzip, winzip, 7-zip을 사용하였으며, 3번에서 사용한 4가지 input을 모두 사용하여 압축률을 비교하였습니다. 압축 프로그램의 세팅은 모두 default로 하였습니다.

Input		hw2		gzip		winzip		7-zip	
File	Size	Size	Ratio	Size	Ratio	Size	Ratio	Size	Ratio
ta_in	1555051	796149	48.8	650898	58.1	626622	59.7	538278	65.4
ps1_in	1985492	931830	53.1	666044	65.5	642504	67.6	509379	74.3
ps2_in	2006559	635400	68.3	512756	74.4	469463	76.6	24244	98.8
ps3_in	2096128	8817	99.6	2171	99.9	2663	99.9	520	100.0

모든 경우에서 다른 압축 프로그램이 직접 짠 LZ78 encoder보다 더 좋은 압축률을 보였습니다. 7-zip을 best compression ratio라고 가정하고 생각하면, best에 비해 15% ~ 20% 정도 압축률이 낮게 나왔습니다. ps2_in의 경우 7-zip이 아주 큰 압축률을 보이는데, 이는 ps2가 100,000byte를 20번 반복하는 패턴을 감지하는데 성공한 것으로 보입니다. 다른 압축 프로그램도 dictionary의 크기를 크게 하면 높은 압축률을 보여줄 것으로 보입니다.

한편 더 좋은 압축률을 얻기 위해서는 작성한 encoder를 조금 더 개선할 필요가 있어 보입니다. 다음은 압축률을 올리기 위해 생각해본 방법들입니다.

1) Dictionary size의 maximum 값을 여러 개를 시도하고, 그 중에 제일 좋은 것을 선택합니다. 압축률은 input text의 형태에 따라 크게 차이가 나기 때문에, 이에 따라 최적인 dictionary size도 다를 것입니다. 여러 size에 대해 시도하고 encoding시 bit를 할당한다면, 더 좋은 압축률을 얻을 수 있을 것입니다.

2) 2-pass encoding을 수행합니다. Dictionary를 encoding할 때, 할당할 bit를 static하게 고정시키는 건 많은 낭비가 있습니다. 따라서 얻은 dictionary에 대해 encoding을 적용하면 더 나은 압축률을 얻을 수 있을 것입니다. 예를 들자면, dictionary를 생성한 후 index에 대해 Huffman encoding을 적용하면, index에 따라서 다른 bit수가 할당되어 encoding되므로 불필요한 bit가 많이 줄어들 것입니다.