Online Compression

LZ77 Algorithm

BY

PRATEEK JAIN (170010007)

GAGAN G.B. (170020029)

Introduction

The LZ77 compression is a lossless data compression method. It exploits the fact that words and phrases within a text file are likely to be repeated. When there is repetition, they can be encoded as a pointer to an earlier occurrence, with the pointer followed by the number of characters to be matched. It is a very simple technique that requires no prior knowledge of the source and seems to require no assumptions about the characteristics of the source.

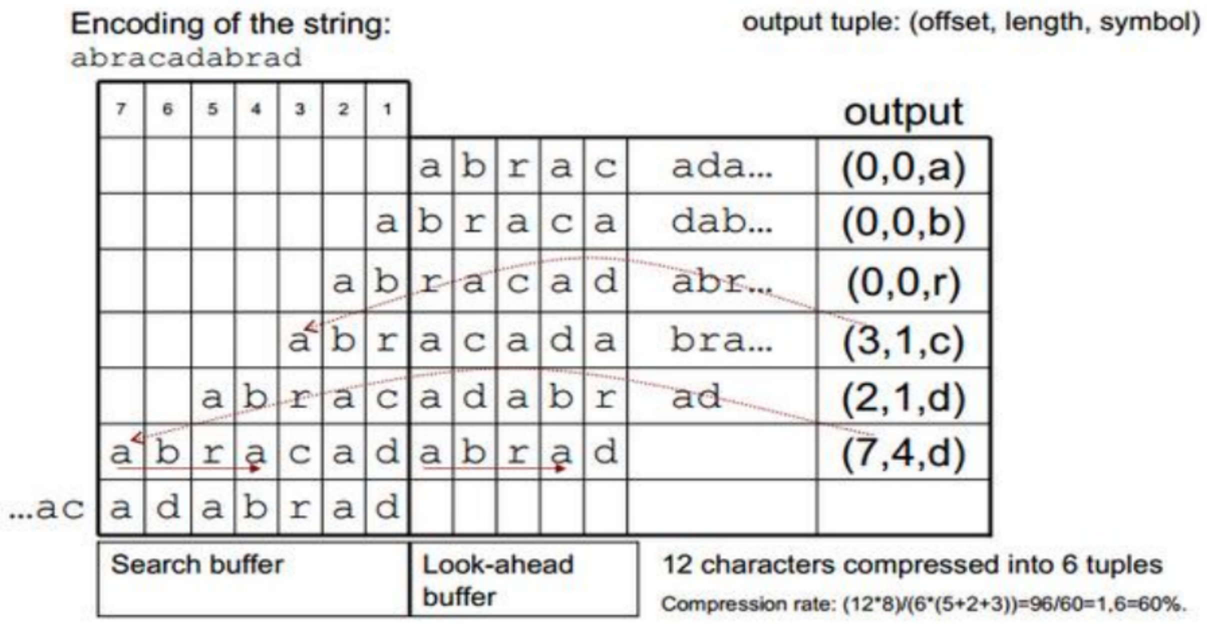
In the LZ77 approach, the dictionary work as a portion of the previously encoded sequence. The encoder examines the input sequence by pressing into service of sliding window which consists of two parts: Search buffer and Look-ahead buffer. A search buffer contains a portion of the recently encoded sequence and a look-ahead buffer contains the next portion of the sequence to be encoded.

The algorithm searches the sliding window for the longest match with the beginning of the look-ahead buffer and outputs a pointer to that match. It is possible that there is no match at all, so the output cannot contain just pointers. In LZ77 the sequence is encoded in the form of a triple , where 'o' stands for an offset to the match, 'l' represents length of the match, and 'c' denotes the next symbol to be encoded. A null pointer is generated as the pointer in case of absence of the match (both the offset and the match length equal to 0) and the first symbol in the look-ahead buffer i.e. (0,0,"character"). The values of an offset to a match and length must be limited to some maximum constants. Moreover the compression performance of LZ77 mainly depends on these values.

Algorithm:

```
While (look-Ahead Buffer is not empty)
{
    Get a pointer (position, length) to longest match;
    if (length > 0)
    {
        Output (position, Longest match length, next symbol );
        Shift the window by (length+1) positions along;
    }
    Else
    {
        Output (0, 0, first symbol in the look-ahead buffer);
        Shift the window by 1 character along;
    }
}
```

Example:



Encoding of the string:
abracadabrad

output tuple: (offset, length, symbol)

	7	6	5	4	3	2	1							output
								a	b	r	a	c	ada...	(0,0,a)
							a	b	r	a	c	a	dab...	(0,0,b)
						a	b	r	a	c	a	d	abr...	(0,0,r)
					a	b	r	a	c	a	d	a	bra...	(3,1,c)
			a	b	r	a	c	a	d	a	b	r	ad	(2,1,d)
	a	b	r	a	c	a	d	a	b	r	a	d		(7,4,d)
...ac	a	d	a	b	r	a	d							

Search buffer | Look-ahead buffer | 12 characters compressed into 6 tuples

Compression rate: (12*8)/(6*(5+2+3))=96/60=1,6=60%.

How to run Code:

To compress:

```
python3 encoder.py <path-to-data-file> <search-buffer-length>
<look-ahead-buffer-length>
```

Uncompressed data will be stored in binary format in “uncompressed.txt”
and Compressed data will be stored in binary format in “compressed.txt” and

To compare compressed and uncompressed file:

```
python3 compare.py
```

To decompress:

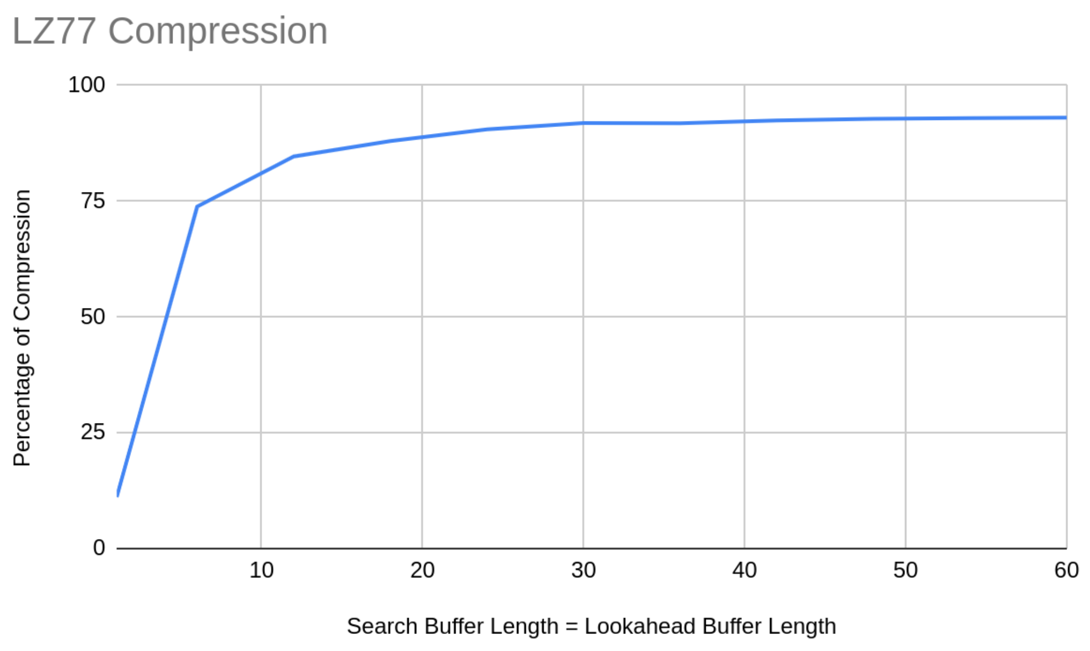
```
python3 decoder.py <path-to-compressed-file> <search-buffer-length>
```

Decompressed data will be stored in “decompressed.txt”

(While decompressing, search buffer length should be greater than or equal to search buffer length used while encoding)

Analysis

We took data such that there is repetition in data after every 6 characters. Then we increased the length of search buffer and lookahead buffer keeping them equal and observed change in compression.



It can be clearly seen that larger the length of search buffer and lookahead buffer, higher is the compression. It is because larger strings from the lookahead buffer can be matched with the larger substrings in the search buffer leading to more compression.