Extended Introduction to Computer Science CS1001.py

Module H (Text): Ziv–Lempel Compression

Instructors: Elhanan Borenstein, Michal Kleinbort
Teaching Assistants: Noam Parzanchevsky, Asaf Cassel,
Shaked Dovrat, Omri Porat

School of Computer Science
Tel-Aviv University
Spring Semester 20-21
http://tau-cs1001-py.wikidot.com

* Slides based on a course designed by Prof. Benny Chor

Ziv–Lempel Compression: Overview

- Lossless text compression
- Exploits text repetitions (as opposed to Huffman).
- Basis of zip, winzip, Tar, and numerous other commercial compression packages.

Ziv–Lempel Compression: History

Proposed by Yaacov Ziv and Abraham Lempel* in a seminal 1977 paper (“A Universal Algorithm for Sequential Data Compression”, IEEE transactions on Information Theory).

Their algorithm went through several modifications and adjustments. The one used most these days is the modification by Terry Welch, in 1984, and known today as LZW compression.

Unlike Huffman, all variants of LZ compression do not assume any knowledge of character distribution. The algorithm finds redundancies in texts using a different strategy.

We will go through this important compression algorithm in detail.

*who later became professors at (correspondingly) the EE and CS faculties, Technion.

Huffman vs. Ziv Lempel: Fundamental Difference

Huffman compression computes character frequencies based on some standard corpus. These frequencies are used to generate encodings for single characters, which are subsequently employed to compress and decompress any future text.

The statistics (or derived dictionaries) are either shared by both sides before communication starts, or have to be explicitly transmitted as part of the communication.

By way of contrast, Ziv-Lempel compression is adaptive: there is no precomputed statistics. The basic redundancies employed here are repetitions in the text, which are quite frequent in human generated (and other forms of) texts.

There is no need here to share any data before transmission commences.

Exploiting Text Repetitions

The basic idea of the Ziv-Lempel algorithm is to “take advantage” of repetitions in order to produce a shorter encoding of the text.

- Let T be an n character long text. In Python’s spirit, we will think of it as T[0:n].
- Suppose we have a k long repetition ($k > 0$) at position j and at position $p = j + m$ ($m > 0$), namely:
  T[j:j+k] = T[p:p+k].

Basic Idea: Instead of encoding T[p:p+k] character by character, we can represent it by identifying the backwards offset m to the starting point of the first occurrence, j, and the length of the repetition, k.

Example

We will see the implementation details later, but let's run some examples:

```
>>> LZW_compress("abcxyzabc")
['a', 'b', 'c', 'x', 'y', 'z', [6, 3]]
```

```
>>> LZW_compress("abcxyzabcxyz")
['a', 'b', 'c', 'x', 'y', 'z', [6, 6]]
```

```
>>> LZW_compress("abcabcabc")
```

???

Example (cont)

We will see the implementation details later, but let’s run some examples:

```
>>> LZW_compress("abcxyzabc")
['a', 'b', 'c', 'x', 'y', 'z', [6, 3]]
```

```
>>> LZW_compress("abcxyzabcxyz")
['a', 'b', 'c', 'x', 'y', 'z', [6, 6]]
```

```
>>> LZW_compress("abcabcabc")
['a', 'b', 'c', [3, 6]]
```

Important: the repetition encoded segment T[j:j+k] may go beyond T[p] (into the “future”).

Example (cont. cont.)

We will see the implementation details later, but let’s run some examples:

```
>>> LZW_decompress(['a', 'b', 'c', [3, 6]])
'abcabcabc'
```

When decompressing pairs such as [3,6] we re-generate the text based on a prefix of itself!

How to Represent Repetitions

Ziv-Lempel advocates looking for repetitions in only a bounded part of the text. The standard recommendation is to consider only a ”window” of the $W = 2^{12} - 1 = 4095$ most recent characters.

Can you think of the pros and cons of this choice?

How to Represent Repetitions (cont.)

This choice has the disadvantage that repeats ”below the horizon”, *i.e.* earlier than 4095 most recent characters, will be ignored.

It has the advantage that $1 \leq m \leq W$ can be represented using a small, fixed number of bits (12 bits for 4095 size window).
Another advantage is that as we compress the text we only need to keep the last 4095 characters, rather than the whole text seen so far (think of generators in this context).

Note: if we choose not to limit the offset m then we'd need $O(logn)$ bits to represent m, where n is the length of the whole text we want to compress. This is because the offset can be as large as $O(n)$.
Another option is to use j iteself (the starting point of the earlier repetition), which suffers from the same disadvantages.

Representing k (repetition length)

We have already mentioned that the size of the window, W , is typically restricted to 4,096-1=4,095. Thus the offset, m, can be represented by a fixed length, 12 bits number.

For similar reasons, the length of the match, k is also limited: $1 \leq k \leq L$, typically $L = 2^5 - 1 = 31$. So the length, k, can be represented by a fixed length, 5 bits number.

High Level LZW Compression

Instead of producing a string consisting of bits right away, we decompose this task into two: An intermediate output (in an intermediate format), and then a final output (bits). The intermediate format will be easier to understand (and to debug, if needed).

So at the first stage, we produce a list, whose elements are either single characters, or pairs *m, k*, where *m* is an offset, and *k* is a match length.

Later at the second stage, we will ”translate” both single uncompressed characters and pairs *m, k* to binary.

Pseudo-Code for LZW Compression (1st stage)

To encode the string T[0:n], using a ”sliding window” of size W and maximal repetition of size L:

```
1. Loop over positions in T, starting with the index p=0
2. While the text was not exhausted
    a. Find longest match for T[p...] starting at T[p-m...] for any
       $1 \leq m \leq W$ .
    b. Suppose this match is of length k: T[p:p+k] == T[p-m:p-m+k]
    c. if $k \leq 1$ (no match found or match is a single char):
        I.  Output the single character T[p]
        II. Move on to next location in text: p = p+1.
    d. else: ($k \geq 2$)
        I.  Output the pair of integers [m, k].
        II. Update the text location: p = p+k.
```

High Level LZW Compression: Improvement

At the second stage we will transform the intermediate format to the final compressed binary string.

Remember:

```
>>> LZW_compress("abcxyzabc")
['a', 'b', 'c', 'x', 'y', 'z', [6, 3]]
```

High Level LZW Compression: Improvement (cont.)

At the second stage we will transform the intermediate format to the final compressed binary string.

Remember:

```
>>> LZW_compress("abcxyzabc")
['a', 'b', 'c', 'x', 'y', 'z', [6, 3]]
```

To distinguish between a single character and an (m,k) entry, a '0' will be placed before a single ASCII character, while a '1' will be placed before an (m,k) entry.

High Level LZW Compression: Improvement (cont. cont.)

Let’s do some arithmetics: we need 12+5 bits for an (m,k) entry in the intermediate format. But in the binary representation we need an extra bit as mentioned above. So we need a total of 1+12+5=18 bits for an (m,k) entry. A single character in ASCII requires 7 bits, and with the extra bit mentioned above we have 1+7 = 8 bits per uncompressed character. Thus, for 2 uncompressed characters we need 16 bits, which is smaller than 18!

Conclusion: Recording repetitions of length 2 is wasteful in terms of bits used vs. bits saved. Thus we will restrict $k > 2$.

Pseudo-Code for LZW Compression (1st stage improved)

To encode the string T[0:n], using a ”sliding window” of size W and maximal repetition of size L:

```
1. Loop over positions in T, starting with the index p=0
2. While the text was not exhausted
    a. Find longest match for T[p...] starting at T[p-m...] for any
       $1 \leq m \leq W$ .
    b. Suppose this match is of length k: T[p:p+k] == T[p-m:p-m+k]
    c. if $k \leq 2$ (no match found or match is ≤ two chars):
        I.  Output the single character T[p]
        II. Move on to next location in text: p = p+1.
    d. else: ($k \geq 3$)
        I.  Output the pair of integers [m, k].
        II. Update the text location: p = p+k.
```

LZW Compression: Implementation

Finding a maximum match quickly is also a major factor determining the time efficiency of the compression algorithm. Hashing and other data structures allow to speed up the computation.

We present a simple iterative procedure for the task, which does not employ any sophisticated data structures. Its performance (both in terms of running time, will not be as good as the optimized, commercial packages. But unlike the packages, you will fully understand what goes on here.

Maximum Match

Our first task is locating the maximum matches.

The function maxmatch returns the offset and the length of a maximum length match T[p:p+k]==T[p-m:p-m+k] within prescribed window size backwards and maximum match size.

The function maxmatch(T,p,W,L) has four arguments:

- T, the text (a string)
- p, an index within the text
- W, a size of window within which matchs are sought
- L, the maximal length of a match that is sought

The last two arguments will have the default values $2^{12} - 1$, $2^{5} - 1$, respectively.

Maximum Match: Python Code

```
def maxmatch(T, p, W=2**12-1, L=2**5-1):
    """ finds a maximum match of length k<=L within a
    W long window, T[p:p+k] = T[p-m:p-m+k].
    Returns m (offset) and k (match length) """
    assert isinstance(T,str)

    n = len(T)
    m = 0
    k = 0
    for offset in range(1, 1+min(p, W)):
        match_len = 0
        j = p-offset #starting point of earlier repetition
        while match_len < min(n-p, L) and \
                T[j+match_len] == T[p+match_len]:
            match_len+=1  # T[j:j+match_len]==T[p:p+match_len]
        if match_len > k:
            k = match_len
            m = offset
    return m, k # returned offset is smallest one (closest to p)
                # among all max matches
```

Maximum Match: A Few Experiments

```
>>> s = "aaabbbaaabbbaaa"

>>> maxmatch(s,0)
(0, 0)
>>> maxmatch(s,1)
(1, 2)
>>> maxmatch(s,2)
(1, 1)
>>> maxmatch(s,3)
(0, 0)
>>> maxmatch(s,4)
(1, 2)
>>> maxmatch(s,5)
(1, 1)
>>> maxmatch(s,6)
(6, 9)
```

Maximum Match: A Few Experiments

```
>>> text = "how much wood would the wood chuck chuck
if the wood chuck would chuck wood should could hood"
>>> maxmatch(text,13)
(5, 3)
>>> maxmatch(text,23)
(15, 6)
>>> maxmatch(text,34)
(6, 7)
```

Next: From Text to Intermediate Format

LZW_compress produces a list, whose elements are either single characters (in case of a repeat of length smaller than 3), or pairs (m, k), where m is an offset, and k is a match length. The default bounds on these numbers are $1 \leq m < 2^{12}$ (12 bits to describe) and $2 \leq k < 2^{5}$ (5 bits to describe).

The LZW compression algorithm scans the input text, character by character. At each position, p, it invokes maxmatch(text,p). If the returned match value, k, is between 0-2, the current character, text[p], is appended to the list. Otherwise, the pair [m,k] is appended.

If a match (m,k) was identified, we advance the location in the text to be examined next from the current p to p+k (why?).

Intermediate Format LZW Compression: Python Code

```
def LZW_compress(text, W=2**12-1, L=2**5-1):
    """ LZW compression of an ascii text. Produces
        a list comprising of either ascii characters
        or pairs [m,k] where 1<=m<=W is an offset and
        3<=k<=L is a match """
    intermediate = []
    n = len(text)
    p = 0
    while p<n:
        m,k = maxmatch(text, p, W, L)
        if k<=2:
            intermediate.append(text[p]) #  a single char
            p+=1
        else: #k>=3
            intermediate.append([m,k]) # compressing 3+ characters
            p+=k
    return intermediate  # a list composed of chars and pairs
```

Intermediate Format LZW DeCompression: Python Code

Of course, compression with no decompression is of little use.

```
def LZW_decompress(intermediate):
    """ LZW decompression from intermediate format to ascii text"""
    text_lst = []
    for i in range(len(intermediate)):
        if type(intermediate[i]) == str:  # char, as opposed to a p
            text_lst.append(intermediate[i])
        else:
            m,k = intermediate[i]
            for j in range(k): # append k times to result
                text_lst.append(text_lst[-m])
                # a fixed offset m "to the left", as result itself
    return "".join(text_lst)
```

Intermediate Format LZW Compression and DeCompression: A Small Example

```
>>> s = "abc"*20
>>> s
'abcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabc'
%>>> 7*len(s) #number of bits without any compression
%420
>>> inter = LZW_compress(s)
>>> inter
['a', 'b', 'c', [3, 31], [3, 26]]
>>> LZW_decompress(inter)
'abcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabc'
```

As demonstrated above, compression could pass the current location: That is, we can have j+k-1 > p in T[j:j+k] = T[p:p+k].

Intermediate Format LZW Compression and DeCompression: Another Small Example

```
>>> s = "how much wood would the wood chuck chuck if
the wood chuck would chuck wood should could hood"

>>> inter = LZW_compress(s)
>>> inter
['h', 'o', 'w', ' ', 'm', 'u', 'c', 'h', ' ', 'w', 'o', 'o', 'd',
[5, 3], 'u', 'l', 'd', ' ', 't', 'h', 'e', [15, 6], 'c', 'h',
'u', 'c', 'k', [6, 7], 'i', 'f', [24, 15], '\n', [45, 6],
[30, 6], [23, 5], 's', 'h', [18, 6], [6, 5], 'h', [18, 3]]

>>> t = LZW_decompress(inter)
>>> s == t
True
```

But in the End It’s Only Bits Out There

```
>>> s = "abc"*20
>>> s
'abcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabc'
>>> inter = LZW_compress(s)
>>> inter
['a', 'b', 'c', [3, 31], [3, 26]]
>>> LZW_decompress(inter)
'abcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabc'
```

Q1: How many bits are needed for the compressed binary string?

Q2: And how many are needed without any compression? For simplicity, assume we deal with only ASCII chars of 7 bits each.

There and Back Again

Let us complete our LZW tour by going from the intermediate format to the compressed string of bits, and vice versa.

From Intermediate Format to Compressed Binary String

To distinguish between a single character and an [m,k] entry, a '0' will be placed before a single ascii character, while a '1' will be placed before an [m,k] entry.

```
def inter_to_bin(intermediate, W=2**12-1, L=2**5-1):
    """ converts intermediate format compressed list
        to a string of bits"""
    W_width = math.floor(math.log(W,2)) + 1
    L_width = math.floor(math.log(L,2)) + 1
    bits = []
    for elem in intermediate:
        if type(elem) == str:
            bits.append("0") #to note a single char ahead
            bits.append((bin(ord(elem))[2:]).zfill(7))
        else: #elem is a list [m,k]
            bits.append("1") #to note a repetition ahead
            m,k = elem
            bits.append((bin(m)[2:]).zfill(W_width))
            bits.append((bin(k)[2:]).zfill(L_width))
    return "".join(ch for ch in bits)
```

Don't forget to import math for the logarithm.

From Compressed Binary String to Intermediate Format

```
def bin_to_inter(bits, W=2**12-1, L=2**5-1):
    """ converts a compressed string of bits
       to intermediate compressed format """
    W_width = math.floor(math.log(W,2)) + 1
    L_width = math.floor(math.log(L,2)) + 1
    inter = []
    n = len(bits)
    p = 0
    while p<n:
        if bits[p] == "0":  # single ascii char ahead (next 7 bits)
            p+=1
            char = chr(int(bits[p:p+7], 2))
            inter.append(char)
            p+=7
        elif bits[p] == "1":  # repeat [m,k] ahead
            p+=1
            m = int(bits[p:p+W_width],2)
            p+=W_width
            k = int(bits[p:p+L_width],2)
            p+=L_width
            inter.append([m,k])
    return inter
```

Don't forget to import math for the logarithm.

The Whole Compress/Decompress Cycle: A Small Example

```
>>> s = "abc"*20
>>> s
'abcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabc'
>>> 7*len(s) #number of bits without any compression
420
>>> inter = LZW_compress(s)
>>> inter
['a', 'b', 'c', [3, 31], [3, 26]]
>>> bits = inter_to_bin(inter)
>>> bits
'011000010110001001100011100000000001111111000000000001111010'
>>> len(binn)
60
>>> len(binn)/(len(s)*7) #60/420
0.14285714285714285
>>> inter2 = bin_to_inter(bits)
>>> inter2==inter #just checking...
True
>>> LZW_decompress(inter2)
'abcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabc'
```

The Whole Compress/Decompress Cycle: Another Example

```
>>> text = "how much wood would the wood chuck chuck if the wood
chuck would chuck wood should could hood"
>>> inter = LZW_compress(text)
>>> bits = inter_to_bin(inter)
>>> bits
'0110100001101111011101110010000001101101011101010110001101101000
0001000000111011101101111011011110110010010000000001010001101110
1010110110001100100001000000111010001101000011001011000000001111
0011001100011011010000111010101100011011010111000000000110001110
1101001011001101000000001100010000100000010110100110100000001100
0100010000000101110001101110011011010001000000010010001101000000
0001100010101101000100000001001000011'
>>> len(bits)
428
>>> len(bits)/(len(text)*7)
0.6574500768049155
```

Time Complexity of the LZW compression

Finding the maximal match for possibly each position is the major consumer of time in our compression procedure.

For any location in the text, p, this function takes up to $W \cdot L$ many operations in the worst case.

For the default parameters, this is $2^{12} \cdot 2^{5} = 2^{17}$ per one position, p. This is a rather pessimistic worst case estimate, as it assumes that for every offset we go all the way to length 31 and do not find a mismatch earlier.

Running maxmatch(T,p) over all text locations will thus take up to 2^{17} times the length of T operations.
This is again a rather pessimistic worst case estimate, as some positions may be skipped over due to repetitions found.

Time Complexity of the LZW Compression (cont.)

Let us assume that one half of the text is skipped over due to repetitions found (a reasonable assumption for human generated text) . If n denotes the text length, compression will require $2^{16} \cdot n$ operations.

Although this is $O(n)$, the ”hidden” constant is very large.

Anyhow, we mentioned already that our implementation is far from being time efficient.

Next, we will test our code for much longer srings than in the toy examples we saw: the New-York Times magazine, and the proteome (set of proteins) of the notorious cholera bacteria.

Additional Examples (time permitting)

For convenience, we “package” together some relevant functions:

```
def process(text):
  """ packages the whole process using LZW compression """
  inter = LZW_compress(text)
  bits = inter_to_bin(inter)
  inter2 = bin_to_inter(bits)
  text2 = LZW_decompress(inter)
  return inter, bits, inter2, text2

def str_to_ascii(text):
  """ Gets rid of non ascii characters in text """
  return ''.join(ch for ch in text if ord(ch)<128)
```

Additional Examples: New-York Times

```
>>> import urllib.request
>>> url = 'http://www.nytimes.com'
>>> nyt = urllib.request.urlopen(url).read().decode('utf-8')
>>> asci_nyt = str_to_ascii(nyt)
>>> inter, bits, inter2, text2 = process(asci_nyt) #a few minutes!
>>> print(text2 == asci_nyt)
True
```

```
>>> len(asci_nyt)*7
896105
>>> len(bits)
329290
>>> 329290 / 896105
0.3674680980465459
```

Compression ratio is about 37% of original text!

There and Back Again: The NY Times Test: Huffman

```
>>> C = generate_hcode(build_huffman_tree(char_count(asci_nyt)))
>>> C
{'p': '00000', 'Z': '000010', 'W': '000011', 'm': '00010',
':': '00011', 'l': '00100', 'o': '00101', '-': '001100',
'k': '0011010', 'R': '0011011', 'M': '001110',
'_': '0011110', 'S': '00111110', 'U': '00111111',
 ...}

>>> hbits = compress(clean_ny_text, C)
>>> len(hbits) / (len(asci_nyt)*7)
0.8300649976463634  # 83% of original
```

It seems that human generated text is more amenable to Ziv-Lempel compression than to Huffman compression.

Compressing the Cholera Proteome

```
>>> cholera = open("Vibrio_cholerae_B33.txt").read()
>>> print(len(cholera),"\n")
3040279
t1 = time.time()
inter, bits, inter2, text2 = process(cholera)
t2 = time.time()
assert text2 == cholera # sanity check

print(t2-t1)
2570.05211019516 # ~43 minutes
>>> print(len(bits)/(7*len(cholera)))
0.7896125886566896  # compression ratio
```

Compression took about 43 minutes!!
The compression ratio is 79%.

Cholera Compression Ratio: Ziv-Lempel vs. Huffman

We saw that the Ziv-Lempel algorithm compresses the Cholera proteome to 79% of its original size. The Cholera proteome is (to the best of our knowledge) not human made. So some properties common in human generated text, like repetitions, are not too frequent. Thus the Ziv-Lempel compression ratio is not very impressive here.

However, most of the cholera proteome text is over the amino acid alphabet, which has just 20 characters. The vast majority of the characters in the text can thus be encoded using under 5 bits on average. This indicates that maybe Huffman could do better here.

Compression: Concluding Remarks

There are additional variants of text compression/decompression algorithms, many of which use combinations of Ziv-Lempel and Huffman encoding. In many cases, it is possible to attain higher compression by employing larger blocks or longer windows.

Our compression algorithm as described so far is greedy: Any repeat of length 3 or more is reported and employed right away. Sometimes this is not optimal: We could have an $[m_1, k_1]$ repeat in position p, and an $[m_2, k_2]$ repeat in position p+1 or p+2, with $k_1 \ll k_2$. Thus a non-greedy algorithm may result in improved compression.

All such improvements would cost more time but produce better compression. In some applications, such tradeoff is well justified.

Compression of gray scale and color images, as well as of documents with a mixture of images and text, uses different approaches. These are based on signal processing techniques, which are often lossy, and are out of scope for our course.

Improvements to LZW: gzip (for reference only)

The gzip variant of LZW was created and distributed (in 1993) by the Gnu[†] Free Software Foundation. It contains a number of improvements that make compression more efficient time wise, and also achieves a higher compression ratio.

As we saw, finding the offset, match pairs [m,k] is the main computational bottleneck in the algorithm. To speed it up, gzip hashes triplets of consecutive characters. When we encounter a new location, p, we look up the entry in the hash table with the three character key T[p]T[p+1]T[p+2]. The value of this key is a set of earlier indices with the same key. We use only these (typically very few) indices to try and extend the match.

[†]The name "GNU" is a recursive acronym for "GNU's Not Unix!"; it is pronounced g-noo, as one syllable with no vowel sound between the g and the n.

Improvements to LZW: gzip (cont.)

To prevent the hash tables from growing too much, the text is chopped to blocks, typically of 64,000 characters. Each block is treated separately, and we initialize the hash table for each.

Hashing improves the running time substantially. To improve compression, gzip further employs Huffman code[‡]. This is used both for the characters and for the offsets (typically close offsets are more frequent than far away ones) and the match lengths.

For every block, the decoding algorithm computes the corresponding Huffman code for all three components (characters, offsets, matches). This code is not known at the receiving end, so the small table descrbing it is sent as part of the compressed text.

[‡]such combination is sometime termed the Deflate compression algorithm.