Computer Science 1001.py

Lecture 21[†]: Introduction to Text Compression

Instructor: Benny Chor

Teaching Assistant (and Python Guru): Rani Hod

School of Computer Science
Tel-Aviv University
Fall Semester, 2011/12
http://tau-cs1001-py.wikidot.com

Lecture 20: Topics

- Rate and minimum distance of codes.
- The volume bound for error correction codes.
- Hamming error correction code.

Lecture 21: Plan

- Intro to text compression.
- Fixed length and variable length codes.
- Prefix free codes.
- Letters' frequencies in natural languages.
- Huffman code: Variable length, letter-by-letter.

Compression (reminder)

In general, a compression scheme consists of two parts: Compression, C, and decompression, D.

Both C and D are functions from binary strings to binary strings. The major goal of a good compression algorithm is to have `len(C(x)) < len(x)`.

This is obviously not hard to achieve. For example, we could simply delete every second bit of `x`. However, under such compression, it is not possible to reconstruct the original string, `x`.

So, in addition to the goal `len(C(x))< len(x)`, we should also satisfy: For every `x`, if we denote `C(x)=y`, then `D(y)=x`.

Universal Lossless Compression is Impossible (reminder)

Claim: There is no universal, lossless compression scheme.

Proof: Simple counting argument. Done on the board last week.

Compression algorithms applied to random text will not compress. They may even expend the text.

The counting argument means we cannot compress everything. But it does not imply we cannot compress a small fraction of all text (or image, audio, or video) sources.

The key to compression are redundancies (*e. g.* repetitions, similarities, etc.). Lossless (and lossy) algorithms look for these and try to exploit them in describing the data more succinctly.

Lossy Compression

When the string in question represent (discretized samples of) analogue data, for example audio, images or video, lossy compression is often used.

If the human eye or ear (at least the average eye or ear) cannot distinguish between the original and the compressed version, then lossy compression is typically acceptable and used. Compression can be achieved, for example, by removing high frequencies from the audio or image, or by reducing the number of color combinations. Video is often compressible in 100:1 ratio, audio in 10:1, and images in 5:1, with hardly any noticeable change in quality.

MP3 for audio, JPEG for images, and MPEG-4 for video are among the well known, lossy compression schemes.

Representation of Characters: ASCII (from lecture 15)

Like everything else in the computer, characters are represented as binary numbers (yet, for convenience, we consider the decimal or hexadecimal representations).

The initial encoding scheme for representing characters is the so called ASCII (American Standard Code for Information Interchange) encoding. It has 128 characters (represented by 7 bits). These include 94 printable characters (English letters, numerals, punctuation marks, math operators), space, and 33 invisible control characters (mostly obsolete).

ASCII is a fixed length code (all characters are encoded by 7 bits).

Representation of Characters: ASCII (from lecture 15)

The initial encoding scheme for representing characters is the so called ASCII (American Standard Code for Information Interchange). It has 128 characters (represented by 7 bits). These include 94 printable characters (English letters, numerals, punctuation marks, math operators), space, and 33 invisible control characters (mostly obsolete).

ASCII Code Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

(table from Wikipedia. 8 rows/16 columns represented in hex, *e.g.* 'a' is 0x61, or 97 in decimal representation)

Huffman Compression

Huffman code translates the binary representation of a text string to a binary string, which should

- Makes it possible to translate back to the original.
- Has a shorter length (usually, not always).
- Work on a letter by letter basis.

The first two properties mean Huffman code is a lossless compression scheme.

Huffman Compression, cont.

Huffman code assigns short coding to letters occurring frequently, and longer coding to letters occurring infrequently.

Thus, on the average, substantial saving can be achieved.

The first step in Huffman coding (and in many other schemes) is to translate the input text to binary (no compression at this step).

ASCII Characters, Represented as Numbers Using `format`

We examine some ways to represent a character as a number:

```
>>> ord("1")       # decimal representation
49
>>> '{:7}'.format(ord("1"))
# decimal representation, 7 place holders, no leading zeroes
'     49'
>>> '{:07}'.format(ord("1"))
# decimal representation, 7 place holders with leading zeroes
'0000049'
>>> '{:07b}'.format(ord("1"))
# binary representation, 7 place holders with leading zeroes
'0110001'
>>> '{:07o}'.format(ord("1"))
# octal representation, 7 place holders with leading zeroes
'0000061'
>>> '{:07x}'.format(ord("1"))
# hexadecimal representation, 7 place holders with leading zeroes
'0000031'
```

For Huffman, we are interested in binary encoding with fixed length 7, and leading zeroes, so `':07b'.format(ord("*"))` is what we want.

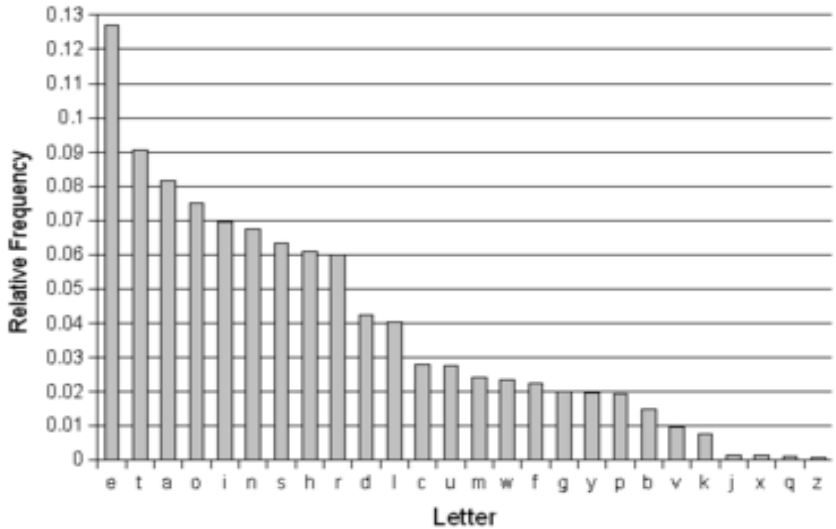
ASCII Strings to Binary Sequences

```
def str_to_bin(text):
  """ translates text to binary reprersentation using
  7 bits per character. Non ascii characters are discarded"""
  return ''.join('{:07b}'.format(ord(ch)) for ch in text
                 if ord(ch)<128)
```

```
>>> text="Geronimo ?????" # ????? is a 5 letter Hebrew word
>>> len(text)
14
>>> str_to_bin(text)
'1000111110010111100101101111111011101101001110110111011110100000'
>>> len(_)
63                # 63 = 7 * (14-5)
```

Frequencies of Letters in Natural Languages

The distribution of single letters in any natural languages' texts is highly non uniform.



We can compute these frequencies by taking a "representative text", or corpus, and simply count letters. For example, in English, "e" appears approximately in 12.5% of all letters, whereas "z" accounts for just 0.1%. In Hebrew, "Yud" (`ord=1497`) appears approximately in 11% of all letters (not counting spaces, digits, etc.) while "Za'in" (`ord=1494`) appears approximately in just 0.8% of all letters.

(As an aside, this observation enables breaking single letter substitution ciphers.)

Huffman Code

In Huffman code, we abandon the paradigm of a fixed length code. Instead, we employ a variable length code, where different letters are encoded by binary strings of different lengths.

Basic Idea:
Frequent letters will be encoded using short binary strings (shorter than 7 bits).
Rare letters will be encoded using long binary strings (typically longer than 7 bits).

This way, the encoding of a typical string will be shorter, since it contains more frequent letters (where we save length) than rare ones (where we pay extra length). (Note that this approach will not work for random strings.)

The rest is just details.
(but, as you surely know, the devil is in the details).

Prefix Free Codes and Fixed Length Codes

In this context, a code is a one-to-one mapping from single characters to binary strings, called codewords.

A code is called fixed length code if all characters are mapped to binary strings of the same length.

A code is called prefix free code if for all pairs of characters γ, τ that are mapped to binary strings $C(\gamma), C(\tau)$, no binary string is a prefix of the other binary string.

As a concrete example, consider the following three codes, both mapping the set of six letters $\{a, b, c, d, e, f\}$ to binary strings.

Code 1:

a	b	c	d	e	f
000	001	010	011	100	101

Code 2:

a	b	c	d	e	f
0	10	110	1110	11110	111110

Code 3:

a	b	c	d	e	f
0	1	00	01	10	11

Prefix Free Codes and Fixed Length Codes

In this context, a code is a one-to-one mapping from single characters to binary strings, called codewords. As a concrete example, consider the following three codes, both mapping the set of six letters $\{a, b, c, d, e, f\}$ to binary strings.

Code 1:

a	b	c	d	e	f
000	001	010	011	100	101

Code 2:

a	b	c	d	e	f
0	10	110	1110	11110	111110

Code 3:

a	b	c	d	e	f
0	1	00	01	10	11

Code 1 is fixed length code. Codes 2 and 3 are not.

Codes 1 and 2 are prefix free codes: No codeword is a prefix of another. Code 3 is not. For example, 1 is a prefix of 11.

Prefix Free Codes and Ambiguity

Code 1:

a	b	c	d	e	f
000	001	010	011	100	101

Code 2:

a	b	c	d	e	f
0	10	110	1110	11110	111110

Code 3:

a	b	c	d	e	f
0	1	00	01	10	11

Code 1 is a fixed length code, while Code 2 and 3 are variable length codes.

Codes 1 and 2 are prefix free codes: No codeword is a prefix of another. Code 3 is not. For example, 1 is a prefix of 11.

Why do we care? Suppose you get the binary string 100 (no spaces!). How would you decode it, according to each of the three codes?
100 encodes e by Code 1. It encodes ba by Code 2.

But by Code 3, it could encode baa or bc or ea, so we got ambiguity here, which is bad. This is why we insist on prefix free codes.

Huffman Code: Specification

The input is a set of n characters $\Sigma = \{a_1, a_2, \ldots, a_n\}$, and a set of non-negative weights (counts), $W = \{w_1, w_2, \ldots, w_n\}$, corresponding to the characters.

The output is a variable length, prefix free, code, $C = Code(\Sigma, W) = \{c_1, c_2, \ldots, c_n\}$, where each c_i is a binary string.

The weighted length of a code $D = \{d_1, d_2, \ldots, d_n\}$ with respect to the set of weights $W = \{w_1, w_2, \ldots, w_n\}$ is defined as $L_W(D) = \sum_{i=1}^{n} w_i \cdot \text{len}(d_i)$.

Goal:
Construct a code, C, so that for any code, D, $L_W(C) \leq L_W(D)$.

Huffman Code: Construction

- Collect letters' statistics from text.
- Use a priority queue (list) to represent letters' counts.
- Iterate the following: Fetch the two smallest item from priority queue. Join them to a new, compound item, whose weight equals the sum of the two weights.
- The end result is a priority queue (list) with two compound items.
- It implicitly represents a binary tree.
- The leaves of this tree are individual characters.
- Representing left by `0`, and right by `1`, we construct the Hufmann encoding.

Counting Characters

```
def char_count(text):
    """ counts the number of occurances of ascii characters -
    ord(ch)<128, in a text. Returns a dictionary, with
    keys being the observed characters """
    d = {}
    for ch in text:
        if ord(ch)>=128: continue
        d[ch] = 1+d.get(ch, 0)
    return d

>>> char_count("aaaaaaaaaaaabbc  7 111 ? <>+")
{'a': 12, ' ': 5, 'c': 1, 'b': 2, '+': 1, '1': 3,
'7': 1, '<': 1, '>': 1}

>>> char_count("abcdefghijklmnopqrstuvwxyz11111111111111111")
{'1': 17, 'a': 1, 'c': 1, 'b': 1, 'e': 1, 'd': 1, 'g': 1, 'f': 1,
'i': 1,'h': 1, 'k': 1, 'j': 1, 'm': 1, 'l': 1, 'o': 1, 'n': 1,
'q': 1, 'p': 1, 's': 1, 'r': 1, 'u': 1, 't': 1, 'w': 1, 'v': 1,
'y': 1, 'x': 1, 'z': 1}
```

Counting Characters: A Real Example

Of course the previous two examples are very far from representing any real letters' distribution. In the spirit of Gödel self referential statements, it is appropriate we collect letters' frequencies from Wikipedia's own Huffman's code entry.

```
>>> import urllib.request
>>> req = urllib.request.Request \
      (url='http://en.wikipedia.org/wiki/Huffman_coding',
          headers={'User-Agent': "Mozilla/5.0"})
>>> btext = urllib.request.urlopen(req).read()
>>> type(btext)
<class 'bytes'>
>>> wikitext = btext.decode("utf8") ; type(wikitext)# 2 in one line
<class 'str'>
>>> len(wikitext)
96837    # number of characters
>>> wikitext[0:35]; wikitext[220:270]
'<!DOCTYPE html PUBLIC "-//W3C//DTD '
'Huffman coding - Wikipedia, the free encyclopedia<'
```

This rather mysterious sequence of commands first convinces Wikipedia to download the relevant page, then transforms it from Bytes to unicode (utf8).

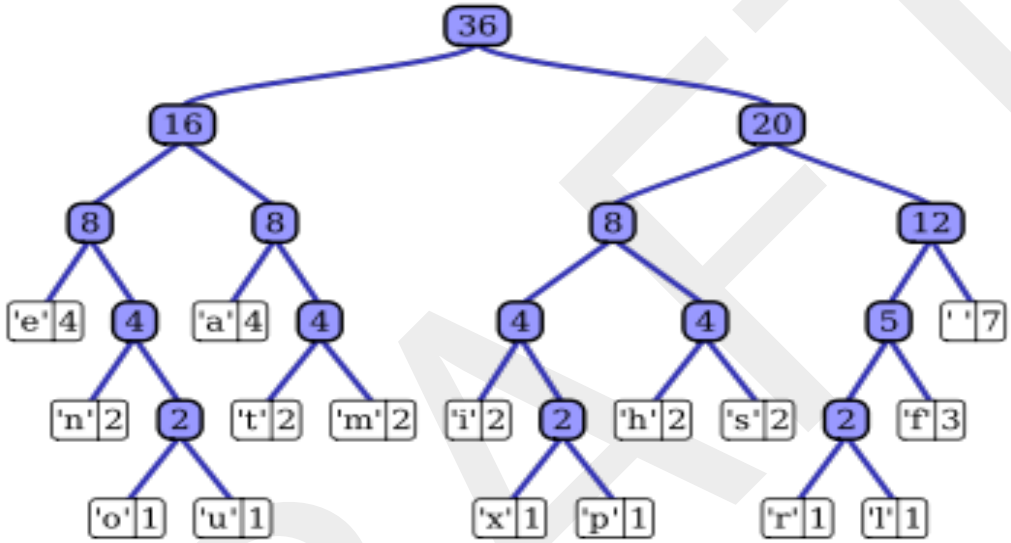
Counting Characters: Real Results

```
>>> len(wikitext)
96837     # number of characters
>>> wikicounts=char_count(wikitext)
>>> wikiqueue = sorted(wikicounts.items(), key=lambda x:x[1])
# sorting the counts dictionary into a priority queue, using
# the second entry (counts) as the sorting key
>>> wikiqueue
[('~',2),('^',3),('Y',4),('X',6),('Q',10),('*',11),('+',12),
('K',13),('U',18),('J',20),('G',31),('O',39),('j',44),('V',45),
("'",46),('[',50),(']',50),('?',53),('Z',56),('{',57),('}',57),
('N',58),('z',58),('!',69),('R',73),('M',74),('\\',85),('F',101),
('I',102),('q',105),('4',115),('W',120),('P',122),('B',136),
('#',141),('(',142),(')',142),('L',145),('7',145),('T',147),
('D',148),('S',150),('6',158),('5',161),('9',183),('3',193),
('&',194),('E',204),('8',223),('A',230),('C',251),('H',260),
('2',309),('%',405),(',',406),('x',464),('1',480),('_',563),
(';',575),('v',627),('0',629),(':',643),('.',663),('\t',688),
('k',770),('y',835),('\n',920),('-',930),('b',1065),('w',1212),
('u',1334),('g',1424),('=',1729),('f',1842),('m',1849),('p',2085),
('h',2212),('c',2508),('/',2544),('<',2752),('>',2752),('d',2849),
('r',3328),('l',3480),('"',3540),('o',3918),('s',3963),('n',4179),
('a',5254),('t',5893),('i',6216),('e',6824),(' ',6938)]
```

Building the Huffman Code: A Road Map

- Preprocessing
  - Collect character counts from a representative corpus.
  - Sort entries by counts.
  - Form a priority queue.
  - Iteratively build the Huffman tree from this priority queue.
- Turn the tree into an encoding dictionary (ascii characters to binary strings).
- Reverse the dictionary to get a decoding dictionary (binary strings to ascii characters).

Building the Huffman Tree: White/Black Board Demo



Huffman tree generated from the character counts of the text "this is an example of a huffman tree" (text and tree from Wikipedia).

```
>>> count=char_count("this is an example of a huffman tree")
>>> sorted(count.items(),key=lambda x:x[1])
[('l', 1), ('o', 1), ('p', 1), ('r', 1), ('u', 1), ('x', 1),
('i', 2), ('h', 2), ('m', 2), ('n', 2), ('s', 2), ('t', 2),
('f', 3), ('a', 4), ('e', 4), (' ', 7)]
```

Reminder: List Operations pop and insert

```
>>> lst=[x for x in range(10,20)]
>>> lst
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> id(lst)
4355065544
>>> lst.pop(0)
10
>>> lst
[11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> id(lst)
4355065544
>>> lst.insert(6,"a")
>>> lst
[11, 12, 13, 14, 15, 16, 'a', 17, 18, 19]
>>> id(lst)
4355065544
```

Both operations mutate the list (and do not change its identity).

The bisect Module

Suppose you have a sorted list, lst, and you wish to insert an additional item in right place, so the mutated list will be sorted. You could just insert the additional item somewhere, and then sort the resulting list. Sorting an n-element list takes $O(n \log n)$ operations, and will be an overkill (especially for long lists).

A better solution would be to find the position where the additional item should go, using binary search, and then to lst.insert(position,item). Python has a package called bisect a function with the same name, which do this for you.

```
import bisect
>>> lst=[x for x in range(10,20)]
>>> item=14.5
>>> pos=bisect.bisect(lst,item);pos
5
>>> lst.insert(pos,item);lst
[10, 11, 12, 13, 14, 14.5, 15, 16, 17, 18, 19]
>>> item=13
>>> pos=bisect.bisect(lst,item);pos
4
>>> lst.insert(pos,item);lst
[10, 11, 12, 13, 13, 14, 14.5, 15, 16, 17, 18, 19]
```

Building the Huffman Tree

We implement a priority queue by counts of sets of characters, starting with sorted singletons.

```
import bisect
def build_huffman_tree(letter_count):
    """ recieves dictionary with char:count entries
        generates a list of letters representing
        the binary Huffman encoding tree"""
    queue = sorted(letter_count.items(), key=lambda x:x[1])
    # sort just by the count
    while len(queue) >= 2:
    # combine two smallest elements
        ax, ay = queue.pop(0)   # smallest in queue
        bx, by = queue.pop(0)   # next smallest
        cx = [ax,bx]
        cy = ay+by # combined weight
        just_ys = [y for x,y in queue]
        pos = bisect.bisect(just_ys, cy)
            # Return the index where to insert item cy in
            # list just_ys, assuming it has been sorted.
        queue.insert(pos, (cx,cy))
            # insert to correct position
    return queue[0][0]
    # peels off the irrelevant data (total weight)
```

Building the Huffman Tree: A Small Example

```
>>> small_count=char_count("this is an example of a huffman tree")
>>> sorted(small_count.items(),key=lambda x:x[1])
[('l', 1), ('o', 1), ('p', 1), ('r', 1), ('u', 1), ('x', 1),
('i', 2), ('h', 2), ('m', 2), ('n', 2), ('s', 2), ('t', 2),
('f', 3), ('a', 4), ('e', 4), (' ', 7)]
>>> small_tree=build_huffman_tree(small_count)
>>> small_tree
[[['a','e'],['i','h'],['m','n']],
[[['s','t'],['l','o'],['p','r']],[[['u','x'],'f'],' ']]]
```

It is indeed not obvious to see what is going on here.

Some parsing of this list/tree may help:

Building the Huffman Tree: A Small Example, cont.

It is indeed not obvious to see what is going on here.

Some parsing of this list/tree may help:

```
>>> len(small_tree)
2   #   as expected
>>> small_tree[0]
[['a','e'],['i','h'],['m','n']]
>>> small_tree[0][0]
['a','e']
>>> small_tree[0][0][0]
'a'   # 'a' will be encoded by 000
>>> small_tree[0][1]
[['i', 'h'], ['m', 'n']]
>>> small_tree[0][1][1]
['m', 'n']
>>> small_tree[0][1][1][1]
'n'   # 'n' will be encoded by 0111
>>> small_tree[1]
[[['s','t'],['l','o'],['p','r']],[['u','x'],'f'],' ']]
>>> small_tree[1][1]
[['u','x'],'f'],' ']
>>> small_tree[1][1][1]
' '   # ' ' will be encoded by 111
```

Observe that indeed more frequent letters get shorter encodings.

Building the Huffman Tree: A Somewhat Large Example

```
>>> wikicounts=char_count(wikitext)   # cont. from before
>>> wiki_huff=build_huffman_tree(wikicounts)
>>> wiki_huff
[[[['n',[['b',['_',';']],'h']],[[[[[')','L'],
['7',[[[['X',['Y',['~','^']]],'U'],'0'],'R']]],
[['T','D'],['S','6']]],'w'],'c'],[[['v','0'],
[['2',[['M','\\'],'5']],':']],'/']]],[['a',[['u',
['.','\t']],'<']],[['>','d'],'t']]],[[['i',[['g',
[[[[[['J',['Q','*']],'j'],['V',"'"]],'9'],
['3','&']],'k']],'r']],['e',[[[['%',[[['[',']'],
'F'],'E']],'y'],'='],'1']]],[[' ',['"',[[[',',
[['I','q'],[['?','Z'],[[['+','K'],'G'],'{']]]],
[['8','A'],'x']],'f']]],[[['m',['\n','-']],'o'],
['s',[[[[['4',['}','N']],['W','P']],'1'],[['C','H'],
[[['z','!'],'B'],['#','(']]]],'p']]]]]]
```

Again, it is not obvious to see what is going on here.