Class Notes CS 3137

1 LZW Encoding

References: Data Structures & Their Algorithms, Harper Collins Publishers, Harry R. Lewis and Larry Denenberg, 1991, and Data Structures and Algorithms, A. Drozdek, Brooks/Cole 2001.

A key to file data compression is to have repetitive patterns of data so that patterns seen once, can then be encoded into a compact code symbol, which is then used to represent the pattern whenever it reappears in the file.

For example, in images, consecutive scan lines (rows) of the image may be indentical. They can then be encoded with a simple code character that represents the lines. In text processing, repetitive words, phrases, and sentences may also be recognized and represented as a code.

A typical file data compression algorithm is known as LZW - Lempel, Ziv, Welch encoding. Variants of this algorithm are used in many file compression schemes such as GIF files etc. These are *lossless* compression algorithms in which no data is lost, and the original file can be entirely reconstructed from the encoded message file.

The LZW algorithm is a greedy algorithm in that it tries to recognize increasingly longer and longer phrases that are repetitive, and encode them. Each phrase is defined to have a prefix that is equal to a previously encoded phrase plus one additional character in the alphabet. Note "alphabet" means the set of legal characters in the file. For a normal text file, this is the ascii character set. For a gray level image with 256 gray levels, it is an 8 bit number that represents the pixel's gray level.

In many texts certain sequences of characters occur with high frequency. In English, for example, the word *the* occurs more often than any other sequence of three letters, with *and*, *ion*, and *ing* close behind. If we include the space character, there are other very common sequences, including longer ones like *of the*. Although it is impossible to improve on Huffman encoding with any method that assigns a fixed encoding to each character, we can do better by encoding entire **sequences** of characters with just a few bits. The method of this section takes advantage of frequently occurring character sequences of any length. It typically produces an even smaller representation than is possible with Huffman trees, and unlike basic Huffman encoding it 1) reads through the text only once and 2) requires no extra space for overhead in the compressed representation.

The algorithm makes use of a *dictionary* that stores character sequences chosen dynamically from the text. With each character sequence the dictionary associates a number; if s is a character sequence, we use $codeword(s)$ to denote the number assigned to s by the dictionary. The number $codeword(s)$ is called the code or code number of s. All codes have the same length in bits; a typical code size is twelve bits, which permits a maximum dictionary size of $2^{12} = 4096$ character sequences.

The dictionary is initialized with all possible one-character sequences, that is, the elements of the text alphabet (assume N symbols in the alphabet) are assigned the code numbers 0 through N-1 and all other code numbers are initially unassigned. The text w is encoded using a greedy heuristic: at each step, determine the longest prefix p of w that is in the dictionary, output the code number of p, and remove p from the front of w; call p the current match. At each step we also modify the dictionary by adding a new string and assigning it the next unused code number. The string to be added consists of the current match concatenated to the first character of the remainder of w. It turns out to be simpler to wait until the next step to add this string; that is, at each step we determine the current match, then add to the dictionary the match from the previous step concatenated to the first character of the current match. No string is added to the dictionary in the very first step.

The table on the last page shows the encoder algorithm (LZWcompress) applied to the string:
a a b a b a c b a a c b a a d a a

```
LZWcompress()
    enter all letters to the table;
    initialize string s to the first letter from input;
    while any input left
        read character c;
        if s+c is in the table
            s = s+c;
        else output codeword(s);
            enter s+c to the table;
            s = c;
    output codeword(s);
```

```
LZWdecompress()
    enter all letters to the table;
    read priorcodeword and output one character corresponding to it;
    while codewords are still left
        read codeword;
        if codeword is not in the table // special case: c+s+c+s+c, also if s is null;
            enter in table string(priorcodeword) + firstchar(string(priorcodeword));
            output string(priorcodeword) + firstchar(string(priorcodeword));
        else enter in table string(priorcodeword) + firstchar(string(codeword));
            output string(codeword);
        priorcodeword = codeword;
```

Figure 1: LZW compress and decompress algorithms

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
	Input:	*a*	*a*	*b*	*a*	*b*	*a*	*c*	*b*	*a*	*a*	*c*	*b*	*a*	*a*	*d*	*a*	*a*	*a*
Code	Output:	1	1	2	6		1	3	7		9		11			4	5		1

iteration	String s	Char c		Coded Output
1	a	a	add aa	1
2	a	b	add ab	1
3	b	a	add ba	2
4	a	b		
5	ab	a	add aba	6
6	a	c	add ac	1
7	c	b	add cb	3
8	b	a		
9	ba	a	add baa	7
10	a	c		
11	ac	b	add acb	9
12	b	a		
13	ba	a		
14	baa	d	add baad	11
15	d	a	add da	4
16	a	a		
17	aa	a	add aaa	5
18	a			1

Code Table

Code	String
1	a
2	b
3	c
4	d
5	aa
6	ab
7	ba
8	aba
9	ac
10	cb
11	baa
12	acb
13	baad
14	da
15	aaa

Algorithm

```
LZWcompress()
        enter all letters in table
        initialize string s to first letter of input
        while any input left
                read character c
                if s+c is in the table
                        s=s+c
                else    output codeword(s)
                        enter s+c in the table
                        s=c
        output codeword(s)
```

LZW Decompress Example

Orig. String	**a**	**a**	**b**	**ab**	**a**	**c**	**ba**	**ac**	**baa**	**d**	**aa**	**a**
Codewords	**1**	**1**	**2**	**6**	**1**	**3**	**7**	**9**	**11**	**4**	**5**	**1**

Prior CodeWord	CodeWord	Output	NewCode Entry
1		a	
1	1	a	5
1	2	b	6
2	6	ab	7
6	1	a	8
1	3	c	9
3	7	ba	10
7	9	ac	11
9	11	baa	12
11	4	d	13
4	5	aa	14
5	1	a	15

Code	String
1	a
2	b
3	c
4	d
5	aa
6	ab
7	ba
8	aba
9	ac
10	cb
11	baa
12	acb
13	baad
14	da
15	aaa

```
LZWdecompress()
    enter all letters to the table;
    read priorcodeword and output one character corresponding to it;
    while codewords are still left
        read codeword;
        if codeword is not in the table // special case: c+s+c+s+c, also if s is null;
            enter in table string(priorcodeword) + firstchar(string(priorcodeword));
            output string(priorcodeword) + firstchar(string(priorcodeword));
        else enter in table string(priorcodeword) + firstchar(string(codeword));
            output string(codeword);
        priorcodeword = codeword;
```