Dictionary Methods

In these types the basis of encoding is again symbols, usually letters, but no statistical inference is used.
The simplistic method, involves a word dictionary, at both ends, with the pointer inside the dictionary being sent. Assuming a dictionary of 500,000 words, a pointer needs to have 20 bits. Assuming a word has 5 characters and each an ASCII code of 8 bits , a 40 character sequence has been coded by a 20 bit sequence, which is quite efficient.
However this static method runs into trouble if words are not found..

An adaptive method can be used, starting from a small dictionary, building up the dictionary on both sides, using the same strategy as for adaptive Huffman or arithmetic codes, with pointers sent for known words, and the words themselves sent when they have not been found.
The decoder reads its input file. Determine whether the current item is a token or uncompressed data. Use tokens to obtain the data from the dictionary, and output the final uncompressed data. Still requires a dictionary of considerable size at both ends

Ziv-Lempel (LZ) Methods

In this case the dictionary based on words is forgotten. The “dictionary” is part of the previously seen input file, moving from right to left through a window, (a sliding window). The window is divided into two parts.
The part on the left is called the *search buffer* and is used as the basis for matching strings of characters. The right side is called the *lookahead buffer* and contains text yet to be encoded.
The encoding is as follows:
Step 1 The present start letter in the lookahead buffer is used to search, trying to match it to the first corresponding letter in the search buffer. If it is found the relative position (offset) is remembered
Step 2 Then moving to the right on both sides an attempt is made to see whether there are further similar characters. If there are this gives rise to a length
Step 3 Process repeated moving in further into the search buffer looking for more matches
Step 4 The longest match, or the last match when lengths are the same, are then built into a token, to which the next letter in the lookahead buffer is also added.
This results in a token made up of three parts

Offset, length, and next symbol in the lookahead buffer

Initially there is no compression, but eventually the file starts compressing provided a matching rate of around 30% is found.

Example: Use the LZ77 with a sliding window having 16 characters as a search buffer and eight characters as the lookahead buffer.

The_cat_ sat_on_the_mat_hoping_to_catch_a_moth_on_the_mattress_before …

	The_cat_	sat	0, 0, “T”
T	he_cat_s	at	0, 0, “h”
Th	e_cat_sa	t	0, 0, “e”
The	_cat_sat	_	0, 0, “_”
The_	cat_sat_	on	0, 0, “c”
The_c	at_sat_o	n	0, 0, “a”
The_ca	t_sat_on	_	0, 0, “t”
The_cat	_sat_on_	t	4, 1, “s”
The_cat_s	at_on_th	e_	4, 3, “o”
The_cat_sat_o	n_the_ma	t	0, 0, “n”

What is the compression, so far, assuming 4 bits for the offset, 3 bits for the length, and 8 bits for an ASCII character?

Improvements

1. Sliding Window – Instead of moving the characters up in an array, a circular queue is used, updating only the pointers to head and end of queue., moving out automatically data and inserting the data at the end, but not physically moving the data.
2. LZSS This improves on L77. It holds the lookahead buffer in a circular queue; the search buffer (the dictionary) in a binary search tree; it creates tokens with two fields instead of three..

The binary search tree is based on the sorting order of the string. All the possible L-size strings in the M-size search buffer are prepared and put into a binary search tree. Each L-size string also has its offset relative to current head of the lookahead buffer. The string in the lookahead buffer is compared to those in the binary tree, looking for a match of the first character, and eventually subsequent characters. The match is output as a token. In this case, instead of [0, 0, “X”] only the uncompressed symbol is output, if no match is found.. To distinguish between tokens and uncompressed data, each is preceded by a bit flag.
After a match, the characters are moved, a new set of L-size strings are built from the current M-size search buffer, and the process is repeated.
While the size of the tree does not change, the shape may change significantly as the strings are put inb and taken out.

Problems: Both of the methods above rely on the search buffer, and the sliding window, so that in practice the matching of strings and words may not be efficient as they may have moved out of the buffer before the current potential match.

3. LZ78

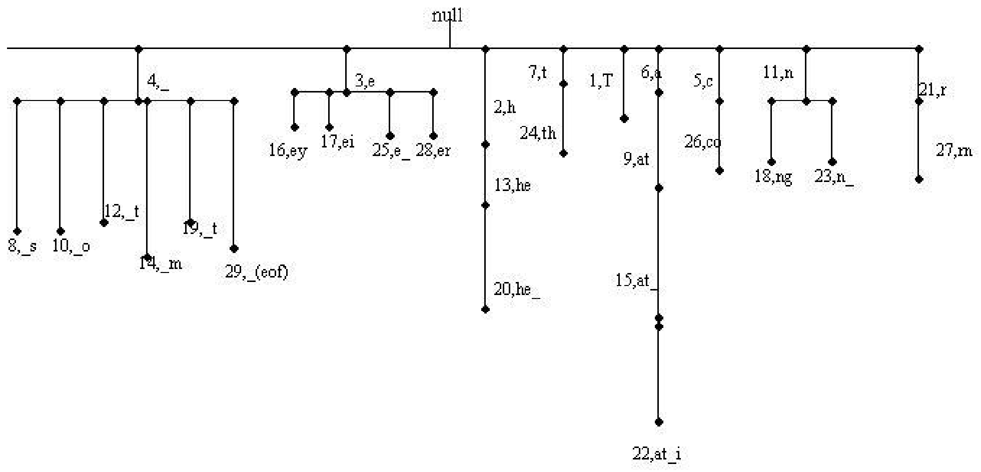
This uses a dictionary of previously encountered strings, building up from an empty dictionary, and limited only by the size of available memory.

It stores the position of the token and the next character, (ie the character that did not match) into a series of growing strings. The most effective data structure is a B+ data structure, starting from the null string, with all possible characters being potential siblings of a parent node. The depth of the tree is the number of characters in a string.

Example: The cat sat on the mat eyeing the rat in the corner.

Table

0	null				
1	T	0, T	16	ey	3, y
2	h	0, h	17	ei	3, i
3	e	0, e	18	ng	11, g
4	_	0, _	19	_t	4, t
5	c	0, c	20	he_	13, _
6	a	0, a	21	r	0, r
7	t	0, t	22	at_I	15, i
8	s	4, s	23	n_	18, _
9	at	6, t	24	th	7, h
10	o	4, o	25	e_	3, _
11	n	0, n	26	co	5, o
12	_t	4, t	27	rn	21, n
13	he	2, e	28	er	3, r
14	_m	4, m	29	_(eof)	4, (eof)
15	at_	9, _			



B+ Tree for the LZ78

4. LZW

This is a variant introduced by Terry Welch on the LZ77. In the LZW method the dictionary is initialized to all the symbols of the alphabet. In the case of 8-bit symbols the first 256 entries of the dictionary are the symbols. Because the dictionary is initialized, the next input symbol is always in the dictionary. Therefore an LZW token consists only of a pointer and does not need to contain a symbol code as in LZ77.
The algorithm is based on the principle that the encoder inputs symbols one by one and accumulates them in a string I. As long as I is in the dictionary a next symbol is added, until Ix is not in the dictionary. At this point the encoder:

(1) outputs the pointer of I;
(2) saves string Ix in the next available dictionary entry
(3) re-initializes string I using symbol x

Example