|  |  |
| --- | --- |
| Aim: | Implement LZ78 for text compression |
| Apparatus: | MATLAB |
| Prerequisit: | Compression basics, Dictionary methods overview |
| Theory: | The LZ78 method does not use any search buffer, look-ahead buffer, or sliding window like LZ77. Instead, there is a dictionary of previously encountered strings. This dictionary starts empty and its size is limited only by the amount of available memory. The encoder outputs two-field tokens. The first field is a pointer to the dictionary; the second is the code of a symbol. Tokens do not contain the length of a string, since this is implied in the dictionary. Each token corresponds to a string of input symbols, and that string is added to the dictionary after the token is written on the compressed stream. Nothing is ever deleted from the dictionary. The dictionary starts with the null string at position zero. As symbols are input and encoded, strings are added to the dictionary at positions 1, 2, and so on. When the next symbol x is read from the input stream, the dictionary is searched for an entry with the one-symbol string x. If none are found, x is added to the next available position in the dictionary, and the token (0, x) is output. This token indicates the string “null x” . If an entry with x is found (at position 37, say), the next symbol y is read, and the dictionary is searched for an entry containing the two-symbol string xy. If none are found, then string xy is added to the next available position in the dictionary, and the token (37, y) is output. This token indicates the string xy, since 37 is the dictionary position of string x. The process continues until the end of the input stream is reached. In general, the current symbol is read and becomes a one-symbol string. The encoder then tries to find it in the dictionary. If the symbol is found in the dictionary, the next symbol is read and concatenated with the first to form a two-symbol string that the encoder then tries to locate in the dictionary. As long as those strings are found in the dictionary, more symbols are read and concatenated to the string. At a certain point the string is not found in the dictionary, so the encoder adds it to the dictionary and outputs a token with the last dictionary match as its first field, and the last symbol of the string (the one that caused the search to fail) as its second field. Table 1 shows the first 14 steps in encoding the string  “sirsideastmaneasilyteasesseasickseals”.  Dictionary Token Dictionary Token   |  |  | | --- | --- | | 1 “s” (0,“s”) | 8 “a” (0,“a”) | | 2 “i” (0,“i”) | 9 “st” (1,“t”) | | 3 “r” (0,”r”) | 10 “m” (0,“m”) | | 4 “” (0,“”) | 11 “an” (8,“n”) | | 5 “si” (1,“i”) | 12 “ea” (7,“a”) | | 6 “d” (0,“d”) | 13 “sil” (5,“l”) | | 7 “ e” (4,“e”) | 14 “y” (0,“y”) |   Table 1: First 14 Encoding Steps in LZ78  In each step, the string added to the dictionary is the one being encoded, minus its last symbol. In a typical compression run, the dictionary starts with short strings, but as more text is being input and processed, longer and longer strings are added to it. The size of the dictionary can either be fixed or may be determined by the size of the available memory each time the LZ78 compression program is executed. A large dictionary may contain more strings and thus allow for longer matches, but the trade off is longer pointers (and thus bigger tokens) and slower dictionary search. |
| Procedure: | 1. Input the string  2. Start from first character ; match the maximum length match in the dictionary.  3. The index of the max length match and the next character form the token.  4. Update the new pattern in the dictionary.  5. Repeat till the end of the string. |
| Results and Conclusion |  |