|  |  |  |  |
| --- | --- | --- | --- |
| **Course Name:** | **Information Theory and Coding Techniques** | **Semester:** | **V** |
| **Date of Performance:** | **01 / 10 / 2024** | **Batch No.:** | **B - 1** |
| **Faculty Name:** | **Prof. Makarand Kulkarni** | **Roll No.:** | **16014022050** |
| **Faculty Sign & Date:** |  | **Grade/Marks:** | **\_\_\_ / 25** |

**Experiment No: -3**

**Title:** **Dictionary Methods for Data Compression**

|  |
| --- |
| **Aim and Objective of the Experiment:** |
| To implement LZW coding for data compression using MATLAB/Python. |

|  |
| --- |
| **COs to be achieved:** |
| **CO1**: Learn the basic mathematics and concepts of data compression techniques.  **CO2**: Identify and apply compression algorithm for different types of data. |

|  |
| --- |
| **Theory:** |
| LZW is a popular variant of LZ78 developed by Terry Welch in 1984. The main feature  of LZW is that it dominates second field of a token. An LZW token consisting of just a pointer to the dictionary.  1. Dictionary is an array of variable size strings.   1. It starts by initializing dictionary to all symbols in alphabets. 2. Next input characters will always be found in the dictionary. 3. LZW is an adaptive data compression method but it is slow to adapt to its inputs. Since strings in dictionary get only one character longer at a time LZW adds one dictionary per phrase and increments the string by one symbol at a time.   **Advantages:**   * The previous dictionary entry is not deleted when new entry is being filled. * Length of match is not taken. Only dictionary address is present so token length is lesser.   **Disadvantages:**   * It is slow to adapt to its input since strings in dictionary get only one character longer at a time. * Since dictionary entry is not deleted memory is more. |

|  |
| --- |
| **Stepwise-Procedure:** |
| **Steps of algorithm:**   1. Define a variable string length array ‘I’ 2. Read input symbols and one by one accumulate them in variable string ‘I’. 3. After each symbol is input and is connected to ‘I’, dictionary is reached for the match up of ‘I’ 4. When ‘I’ is found index is read out as output. 5. When ‘I’ is not found, small x is concatenated to ‘I’ and after these following steps are performed:    1. Encode output dictionary pointer of match of ‘I’    2. Save this ‘Ix’ in next available dictionary entry.    3. Initialize ‘I’ to symbol x and continue till all symbols are encoded.   **Design LZW code for the given input sequence.**  **Wabba$wabba$wabba$wabba$woo$woo$woo**  **Write a program to implement LZW code for above sequence:**  def lempel\_ziv\_compression\_fixed(input\_string):      table = []      dictionary = {}      address = 1      i = 0        while i < len(input\_string):          current\_string = input\_string[i]            j = i + 1          while j < len(input\_string) and current\_string in dictionary:              current\_string += input\_string[j]              j += 1          if current\_string in dictionary and j == len(input\_string):              code\_packet = (dictionary[current\_string], '-')          elif len(current\_string) > 1 and current\_string[:-1] in dictionary:              prefix = current\_string[:-1]              new\_char = current\_string[-1]              prefix\_address = dictionary[prefix]              code\_packet = (prefix\_address, new\_char)          else:              code\_packet = (0, current\_string)            table.append({              "Code Packet": code\_packet,              "Address": address,              "Content": current\_string          })            dictionary[current\_string] = address          address += 1            i += len(current\_string)        return table  input\_string = 'abbaabbaababbaaaabaabba'  final\_result = lempel\_ziv\_compression\_fixed(input\_string)  for entry in final\_result:      print(entry) |

|  |
| --- |
| **Observations:** |
| 1. **Test String: abbaabbaababbaaaabaabba**        1. **Test String: Wabba$wabba$wabba$wabba$woo$woo$woo** |

|  |
| --- |
| **Post Lab Subjective/Objective type Questions:** |
| 1. **Explain dictionary techniques for data compression with an example.**    1. Consider the string: ABABABAB.    2. Initially, the dictionary contains the single-character alphabet: A, B.       * As we process the string, we start adding longer sequences to the dictionary.       * A is found in the dictionary, so it's encoded as its dictionary entry, say 0.       * B is also in the dictionary, so it's encoded as 1.       * AB is a new sequence, so it is added to the dictionary, and assigned the next available code, 2.       * The next A is already in the dictionary (code 0).       * The next B is in the dictionary (code 1).       * The sequence ABA is new, so it gets added to the dictionary with code 3.   The process continues, compressing the string into a sequence of shorter codes based on the dictionary.   1. **Explain advantages of LZW over LZ 77 and LZ 78.**    1. LZW (Lempel-Ziv-Welch)       * LZW is a dictionary-based method that builds a dictionary on the fly without needing to explicitly store it, as it is reconstructed during decompression. It uses the entire history of the input data to create new entries in the dictionary.       * Advantage: LZW builds a dictionary dynamically as the data is processed, making it more efficient for compressing data with repeating patterns. It doesn’t require sending separate dictionary entries, as both encoder and decoder build the same dictionary during compression and decompression.    2. b. LZ77 (Lempel-Ziv 77)       * LZ77 uses a sliding window technique. It searches for repeated patterns in a fixed-size window of previous data and replaces them with references to earlier occurrences (offset-length pairs).       * Limitation: LZ77 needs to maintain a sliding window, which limits how far back it can search for patterns. Also, it has to send offset and length data along with the compressed file.    3. c. LZ78 (Lempel-Ziv 78)       * LZ78 builds a dictionary of patterns explicitly, where the dictionary is shared between the encoder and decoder. However, entries are added based on pairs of symbols and dictionary pointers.       * Limitation: LZ78 needs to explicitly transmit both the dictionary index and the new character with each code, which can add overhead and limit compression efficiency in some cases. 2. **Compare static and dynamic dictionary techniques.**   **Definition:**   * **Static Dictionary:** This method uses a predefined dictionary that remains unchanged during the compression and decompression processes. The dictionary is built before compression begins and is shared between the encoder and decoder. * **Dynamic Dictionary:** In contrast, this technique constructs the dictionary dynamically while processing the input data. The dictionary evolves based on the sequences encountered during compression.   **Dictionary Construction:**   * In static dictionary methods, the dictionary is established in advance, and it works best when the input data matches the patterns defined in that static dictionary. * Dynamic dictionary methods build the dictionary on-the-fly, allowing them to adapt to the specific characteristics of the input data as it is being processed.   **Adaptability:**   * Static dictionaries have limited adaptability since they rely on predefined patterns. They may not perform well if the input data doesn’t align with these patterns. * Dynamic dictionaries are highly adaptable, as they can create a tailored dictionary that reflects the specific input, leading to improved compression for diverse datasets.   **Compression Efficiency:**   * Static techniques might not achieve high compression rates if the input data does not align well with the static dictionary’s definitions. * Dynamic techniques typically achieve better compression rates because they can adjust the dictionary based on the input data, optimizing the representation of repeated patterns.   **Overhead:**   * Static dictionary methods have no overhead in transmitting the dictionary since it is predefined and agreed upon by both the encoder and decoder. * Dynamic techniques may involve some overhead due to the need to implicitly or explicitly transmit information about the evolving dictionary during compression and decompression.   **Use Cases:**   * Static dictionary methods are suitable for scenarios where the data structure is known and consistent, such as text with common words (e.g., a predefined dictionary of English words). * Dynamic dictionary methods are ideal for compressing data with unknown or varying patterns, such as multimedia files or general-purpose data. |

|  |
| --- |
| **Conclusion:** |
| In this experiment, I successfully implemented the Lempel-Ziv compression technique to effectively compress the input string while maintaining the integrity of the original data. The results demonstrate the algorithm's ability to identify and encode repeating patterns, leading to efficient data representation and highlighting the potential for real-world applications in data compression. |

|  |
| --- |
| **Signature of faculty in-charge with Date:** |