8.1 Introduction

This chapter proposes a novel implementation of LZW called IKNTN_LZW (Indexed K Nearest Twin Neighbour Lempel Ziv Welch). The major feature of the conservative LZW data compression algorithm implementations is a common practice, they usually use single dictionary, but the MDLW uses multiple dictionaries based on the length of the input pattern and the three major conventional implementations and one proposed implementation which is discussed in chapter V, and enhanced MDLZW and its computational complexity and how each data structure implementations differ from one another is discussed in detail at chapter VI. The VII\textsuperscript{th} chapter of this thesis proposes a novel clustering algorithm called Indexed K Nearest Twin Neighbour (IKNTN) algorithm. The experimentation result of IKNTN is shown better performance when compared to the traditional conventional data structures. So in order to reduce the computation, time is wasted in searching the large-address-space dictionary, instead of using a unique single dictionary. A clustered dictionary set containing several clustered dictionaries with cluster index are used in the compression algorithm. The algorithm combines the good features of IKNTN with LZW to reduce the computational cost or time. Instead of searching in the huge dictionaries, the proposed approach only searches within the particular cluster. The experimental result shows that the new architecture has several advantages. It can be easily implemented, and reduces computation cost, since it no longer needs to search the dictionary recursively as the conventional implementations do. This architecture improves and modifies the features of LZW algorithm in the following ways. First, the Index value of clusters is to be calculated using each pattern and the cluster for the lookup is to be performed, and the cluster based is to be selected on the Index value. For example, if the index value which is obtained using the input pattern indicates the second cluster, then the search is performed only within the second cluster. If the Index indicates the third clusters, the third cluster is opted for the search. In case, if the cluster of the dictionary does not have such pattern, then the pattern is inserted as a new member or a pattern in the same cluster. Finally each single cluster has twin natured patterns. For example, the cluster used in IKNTN_LZW compression algorithm consists of K small clusters, numbered from 1 to K, where K indicates number of clusters, each cluster only has twin elements based on its
width assigned for the Index value calculation, i.e., the first cluster of the dictionary stores only similar homogenous patterns, and the cluster two has only different homogenous pattern when compared with the first clusters, and so on. So when comparing cluster with another cluster in the dictionary, it has heterogeneous elements or patterns, i.e., a new cluster is formed only when the pattern is heterogeneous to the available clusters. These clusters constitute a cluster set. In general, different address space distributions of the cluster for the dictionary D set will present significantly distinct performances of the IKNTN_LZW compression algorithm. However, the optimal distribution strongly depends on the actual input data files. This chapter mainly focuses on how the important data structures and algorithms like Linear array, Binary search tree, Hash table and Binary insertion sort can play important role in the implementation of IKNTN_LZW approach, and how clustered dictionary implementation of LZW can give optimal reduction of computational cost using the above data structures or algorithms. All types of IKNTN_LZW implementations are done and both theoretical and practical evaluations are also done.

8.2 LZW with Indexed K Nearest Twin Neighbour (IKNTN) Algorithm

The proposed IKNTN_LZW algorithm implementation of the algorithm has three phases: i) cluster identification or new cluster creation phase ii) Switching and coding phase and iii) searching and updating phase. In the proposed algorithm, the initial clustering of the dictionary is not possible because initially the dictionary D is empty or the size of the Dictionary is nil i.e., \(|D| = 0\). So the usage of any traditional clustering algorithm is not suitable for the LZW dictionary clustering. Because of this reason the IKNTN clustering algorithm is proposed in the chapter VII is used. In this novel approach, the features of IKNTN clustering algorithm is clubbed with the LZW. For encoding initially the initial cluster or cluster –zero \(C_0\) contains all the possible roots in ascending order (0 to 255). The \(C_0\) is virtual cluster (represented using the dashed line) or imaginary or not implemented in real practice. So no insertion is made in the initial cluster while encoding (compression). In the initial stage, two variables are initialized i.e., \(K\) equal to Zero and \(L\) equal to two, where \(K\) indicates the number of clusters and \(L\) is the margin for identifying the twin element, or cluster identification.
Initialize \( K = 0 \)

2. \( L = 2 \)

3. \( \text{STRING} = \text{get input character} \)

4. \( \text{WHILE there are still input characters} \)

5. \( \{ \)

6. \( \text{CHAR} = \text{get input character} \)

7. \( \text{Index} = \text{Calculate_Index(STRING + CHARACTER, L)} \)

8. \( \text{If the Index is available in the Index Table then} \)

9. \( \text{Get the Position of Cluster and position and store in J} \)

10. \( \text{Act} = \text{SEARCH (STRING + CHARACTER, C_J);} \)

\( \text{// using appropriate data structure and} \)

\( \text{// Search in the C_Jth Cluster in the dictionary D} \)

11. \( \text{Else} \)

12. \( \text{Create a new cluster and assign Index to the newly created cluster} \)

13. \( \text{Act} = \text{False} \)

14. \( \text{Increment K by one and K is set to J.} \)

15. \( \text{End if} \)

16. \( \text{IF Act equal to true then} \)

17. \( \{ \)

18. \( \text{STRING} = \text{STRING + CHAR} \)

19. \( \} \)

20. \( \text{ELSE} \)

21. \( \{ \)

22. \( \text{Output the code for STRING} \)

23. \( \text{Add STRING + CHAR to the string to \ C_Jth cluster of Dictionary D} \)

\( \text{// use appropriate data structure} \)

24. \( \text{STRING} = \text{CHAR} \)

25. \( \} \)

26. \( \} \)

27. \( \text{Function Calculate_Index(X, L)} \)

28. \( \text{return} \sum_{i=1}^{L} \left( \text{ascii} \left( \text{CURVAL}\left(\text{ith character of X}\right) \right) - 1 \right) \ast \left(256^{L-1}\right) + 1 \)

29. \( \text{End function} \)

---

The first phase of the algorithm is cluster identification or new cluster creation phase. In this phase separate table which is maintained is called Index Table. The purpose of the Index Table which shown in the figure 8.2, figure 8.6, figure 8.10 and figure 8.14 for various implementations is to identify in which cluster the Lookup has to be performed. Initially the table does not have any element or the table empty. After encoding the input sequence maximum elements in the Index Table is \( K \) or this indirectly means that \( K \) is the maximum
number of cluster available for the dictionary D after encoding the input sequence X. For executing the first phase of the algorithm the Index value for \( STRING + CHARACTER \) is to be calculated using the formula shown in the figure 8.1. After calculating the Index, the algorithm starts searching in the index table to determine which cluster the search operation or the data structure operation should take place. Initially, when the Index Table is empty, the search fails in the index table, and so the new cluster is created and it inserts the pattern \( STRING + CHARACTER \) into newly created cluster. For example, if the \( STRING + CHARACTER \) is ‘AB’, then the cluster is not available or no cluster is found, in such case a new cluster is created and added ‘AB’ as the element of newly created or the first cluster, the same takes place for the patterns ‘BC’, ‘CA’, and ‘BA’. In such cases, each time algorithm increments the value of \( K \) by one, so total number of cluster at present is one. If the Index value is already available in the Index Table, then the algorithm switches to next phase.

The second phase of the encoding algorithm is searching and coding phase. The variable \( STRING \) is a variable length string i.e., it may have a group of one or more characters with each character being a single byte, and then taking the first byte from the input file, and placing it in the variable \( STRING \). This is followed by the algorithm looping for each additional byte in the input file. Each time a byte is read from the input file. It is stored in the variable \( CHAR \). Each time after storing byte to \( CHAR \) the algorithm identifies the cluster to be searched with the help of Index Table. Then the search is taken only within the cluster. Search done is based on the \( J \) value returned from the Index Table. To determine the availability of concatenation of two variables \( STRING + CHAR \), a code is already assigned. For example if the \( STRING + CHAR \) is ‘ABB’, then with the help of first phase, the cluster is to be identified, the \( J \) value returned from the Index Table is One. So the search takes place only in the first cluster that is immediately followed by the virtual cluster i.e., \( C_1 \), because the \( J \) value is one of the patterns of ‘ABB’. Initially the length of all Clusters after the first insertion of a member is one or 1 where \( i = 1 \) to \( K \). After the final insertion, each cluster size is represented as \( |C_i| \). If the pattern \( STRING + CHAR \) search result is true, then the \( STRING + CHAR \) is assigned to \( STRING \) and the \( CHAR \) is the next character in the input sequence of \( X \), then length of the pattern increases conditionally, and based on the \( J \), the search process again continues in various clusters. This process continues till the search fails or until the longest match is found, if the longest match is found then the index is given as output string. After each unsuccessful search, the third phase begins, in this phase for the \( STRING + CHAR \) a new code is assigned and stored in the cluster where the search fails, and
CHAR is stored into the STRING and the looping is continued. If the pattern STRING+CHAR is reported in the corresponding cluster, then the function returns true, else the pattern is updated as one of the elements in the cluster where the search is failed. The IKNTN_LZW encoding algorithm is shown in the figure 8.1.

8.3 IKNTN_LZW WITH VARIOUS IMPLEMENTATIONS

The length of the input stream X is |X|. The dictionary is represented as D which has the size |D|. Initially the size of the dictionary, i.e., |D|=0. K is the maximum number of clusters in the dictionary, initially K is also zero. This reveals that, initially the number of clusters in the dictionary is zero. C\textsubscript{i} is where i is 1 to K, except virtual cluster, all other clusters are considered for calculating the computational cost. After each unsuccessful search, each dictionary is updated with STRING+CHAR. For example, for searching the pattern ‘BC’, initially the length of the Index is calculated, and then the index of that pattern and the cluster are found and the search is also performed. For performing the search and implementation, different methods can be used. This chapter of the thesis proposes of experimented four types of implementation on IKNTN_LZW, namely:

- IKNTN_LZW with linear array Implementation
- IKNTN_LZW with Binary search Tree (BST) Implementation
- IKNTN_LZW with Chained Hash table Implementation
- IKNTN_LZW with Binary Insertion Sort implementation.

If linear array is used in the search and cluster creation phase for IKNTN_LZW, then initially there is no array found in the memory. When an index table search fails to find cluster, then a new linear array is created. Whenever the algorithm fails to find an Index in the index table, new linear arrays are created. Each array represents different clusters. When a new array is created that also indexed in the Index Table. For example, initially it does not contain any array or cluster in the memory, when searching for the pattern ‘AB’ and its cluster, the index table fails to achieve the search, in such case a new cluster is created using Linear array and the ‘AB’ is added as the first element in the cluster one. Same process will occur for the patterns ‘BC’, ‘CA’ and ‘BA’. Each time the Index Table succeeded to find the cluster then the algorithm locates the Indexed cluster and search is performed only within that cluster. The cluster created using the linear array, the members of the cluster is not in an arranged or in sorted manner, for this reason, searching Linear search is employed in the linear array. The process of clustering and searching, using the linear array is shown in the
After each unsuccessful search in the cluster, the size of the cluster is gradually increased with one. The experimental results are shown in the figures 8.3 to 8.9, the time taken for encoding the files with IKNTN_LZW using Linear Array is shown in table 8.1.

Figure 8.2 Linear Array Implementation of IKNTN_LZW.

Figure 8.3 Experimental Result of Linear array IKNTN_LZW on alphabet.txt

Figure 8.4 Experimental Result of Linear array IKNTN_LZW Xargs.1
Figure 8.5 Minimum and Maximum time required to encode benchmark files using Linear array data structure implemented IKNTN_LZW Encoding.

For the second type of implementation, instead of linear array, the Binary Search Tree (BST) is employed. Initially there is no Tree found in the memory. When an index table search fails to find cluster, then a new BST is created, each time the algorithm fails to find an Index in the index table, then new BST is created. Each BST represents different clusters. When a new BST is created that is indexed in the Index Table. For example, initially it does not contain any BST or clusters in the memory, when searching for the pattern ‘AB’ and its cluster, the index table fails to achieve this search, in such case a new cluster is created using BST and the ‘AB’ is added as the first node in the cluster one. Same process will occur for the patterns ‘BC’, ‘CA’ and ‘BA’ or other words the ‘BC’, ‘CA’ and ‘BA’ are the primary or rote node of the corresponding BST. Whenever the Index Table succeeds to find the cluster, the algorithm locates the Indexed cluster and search is performed only within that cluster. The process of clustering and searching using the BST is shown in figure 8.6, after each unsuccessful search in the cluster the size of the cluster is gradually increased with one automatically, and then the number of nodes in the corresponding tree is also incremented by one. For example, for searching the pattern ‘ABB’ the algorithm switch is the second tree for searching based on the value returns from the Index Table. For the initial search for the pattern ‘ABB’ the algorithm fails to find the pattern in the tree. So the ‘ABB’ is inserted as the right child of AB which is shown in the figure 8.6, and the number of nodes in the tree is incremented by one. Again when the algorithm search is for the same pattern in the second tree, the algorithm returns true and the element is not added to the tree, and the algorithm
continues. The experimental results are shown in the figures 8.7 to 8.9, the time taken for encoding the files with IKNTN_LZW using BST is shown in table 8.1.

![Figure 8.6 BST Implementation of IKNTN_LZW.](image)

Figure 8.6 BST Implementation of IKNTN_LZW.

![Figure 8.7 Experimental Result of BST data structure with IKNTN_LZW on asyouilk.txt](image)

Figure 8.7 Experimental Result of BST data structure with IKNTN_LZW on asyouilk.txt

![Figure 8.8 Experimental Result of BST data structure with IKNTN_LZW on book1](image)

Figure 8.8 Experimental Result of BST data structure with IKNTN_LZW on book1
Figure 8.9 Minimum and Maximum time required to encode benchmark files using BST data structure implemented IKNTN_LZW Encoding.

Table 8.1 Minimum and Maximum time required to encode benchmark files using Linear Array and BST implemented IKNTN_LZW Encoding during the experimentation

| S no | File name   | Linear Array IKNTN_LZW Encoding time t(|X|) | BST INTN_LZW Encoding time t(|X|) |
|------|-------------|---------------------------------------------|----------------------------------|
|      |             | Min(t(|X|)) | Max(t(|X|))                     | Min(t(|X|)) | Max(t(|X|))          |
| 1    | a.txt       | 855552     | 2804384                         | 924869     | 1439468             |
| 2    | aaa.txt     | 1686230769 | 1755359477                     | 8739452759 | 8752077964          |
| 3    | alphabet.txt| 1029635864 | 1079647967                     | 762047925  | 804645294          |
| 4    | asyoulik.txt| 647587658  | 667273457                      | 461786547  | 479140373         |
| 5    | alice29.txt | 826389841  | 888989826                     | 617019554  | 631659473         |
| 6    | bib         | 736461698  | 753983209                     | 541891092  | 553452360         |
| 7    | bible.txt   | 11034892835| 149803403449                  | 25075702560| 25205934021       |
| 8    | book1       | 7654811834 | 7816408950                    | 4381492357 | 4413794250        |
| 9    | book2       | 5347294036 | 5399948898                    | 349648396  | 3471626465        |
| 10   | cp.html     | 173106418  | 182844401                     | 127941919  | 134890495         |
| 11   | E.coli      | 33324941252| 340306507310                  | 26678373332| 2671462725        |
| 12   | Fields.c    | 88124826   | 95901448                      | 65163289   | 68250883          |
| 13   | geo         | 749655392  | 743586258                     | 523120832  | 536880323         |
| 14   | grammar.lsp | 35258264   | 38080819                      | 26279231   | 30516880          |
| 15   | kennedy.xls | 8214455198 | 8311229965                   | 5848568054 | 5878900996       |
| 16   | lcet10.txt  | 3455122726 | 3497972312                    | 2332215519 | 2344328372        |
| 17   | news        | 2665290678 | 278137614                     | 2059379620 | 2091931906        |
| 18   | obj1        | 152532180  | 157646147                     | 123328414  | 128767338         |
| 19   | obj2        | 1795737959 | 1840606206                    | 1349999192 | 1369568314        |
| 20   | paper1      | 347107268  | 368061154                     | 260830113  | 269163148         |
| 21   | paper2      | 538018451  | 560298656                     | 403347458  | 416816442        |
| 22   | paper3      | 306573333  | 312388486                     | 230574732  | 239952410         |
| 23   | paper4      | 102083082  | 106525052                     | 75392400  | 79586422         |
| 24   | paper5      | 92877789   | 97142803                     | 69309619  | 71877464         |
| 25   | paper6      | 256362903  | 269317042                     | 188949667  | 196920668        |
For the third type of implementation, instead of linear array the Binary Search Tree (BST) is employed. Initially there is no Tree found in the memory. When an index table search fails to find cluster, then a new Chained Hash table is created, each time the algorithm fails to find an Index in the index table then new Chained Hash table is created. Each Hash table represents different cluster. When a Hash Table is created that is indexed in the Index Table. For example, initially it does not contain any Hash Table or clusters in the memory. When searching for the pattern ‘AB’ and its cluster, the index table fails to achieve this search. In such case, a new cluster is created using a new Chained Hash Table and the ‘AB’ is added as the first element in the second row of the cluster one (shown in the figure 8.3). Same process will occur for the patterns ‘BC’, ‘CA’ and ‘BA’ or other words the ‘BC’, ‘CA’ and ‘BA’ are the primary element of the corresponding Hash Table. Each time the Index Table succeeds to find the cluster, then the algorithm locates the Indexed cluster and search is performed only within that cluster. For finding the element in the cluster, the hash function is used in the process of clustering and searching, using the chained Hash table is shown in the figure 8.10. After each unsuccessful search in the cluster, the size of the cluster is gradually increased with one automatically, and then the number of elements in the corresponding Hash table is incremented by one. i.e., initially each hash table has only one element. After each unsuccessful search, the algorithm inserts one element to its corresponding table. For example, for the initial search for the pattern ‘ABB’ hash table search fails, because there is no such element found in the second Hash Table, so the algorithm is inserts ‘ABB’ as the third element in the second row of first hash table. After the insertion of ‘ABB’, the members in the table are incremented by one. The experimental results are shown in the figures 8.11 to 8.13, the time taken for encoding the files with IKNTN_LZW using Chained Hash Table is shown in table 8.2.
Figure 8.10 Hash Table Implementation of IKNTN_LZW.

Figure 8.11 Experimental Result of Chained Hash Table data structure with IKNTN_LZW on E.Coli

Figure 8.12 Experimental Result of Chained Hash Table data structure with IKNTN_LZW on Geo
Figure 8.13 Minimum and Maximum time required to encode bench mark files using Chained Hash Table data structure implemented IKNTN_LZW Encoding.

For the Last type of implementation, BIS algorithm is used. Initially there is no Tire found in the memory. When an index table search fails to find cluster, a new tire is created, whenever the algorithm fails to find an Index in the index table, new tire is created. Each tire represents different cluster. When a tire is created that is indexed in the Index Table. For example, initially it does not contain any tire or clusters in the memory, when searching for the pattern ‘AB’ and its cluster, the index table fails to achieve this search, in such case a new cluster is created using a new Tire and the ‘AB’ is added as the first element in the second row of the cluster one which is shown in the figure 8.14. Same process will occur for the patterns ‘BC’, ‘CA’ and ‘BA’ or other words the ‘BC’, ‘CA’ and ‘BA’ are the primary element of the corresponding Tire. Each time the Index Table succeeds to find the cluster, then the algorithm locates the Indexed cluster and search is performed only within that cluster. For finding the element in the cluster, the BIS algorithm is used. The process of clustering and searching using BIS is shown in the figure 8.14. After each unsuccessful search in the cluster, when the size of the cluster is gradually increased with one automatically, the number of elements in the corresponding tire is also incremented by one. i.e., initially each tire has only one element after creating. After an unsuccessful search, the algorithm inserts the element to that corresponding Tire. For example, for the initial search for the pattern ‘ABBBBD’ tire search fails when there no such element found in the second Tire. So the algorithm inserts the ‘ABBBD’ as the Left childe of ‘ABBB’ or root node. After the insertion of the pattern ‘ABBBBB’ the node in the Tire is incremented by one. So the next time
onwards, the algorithm never inserts the same pattern in any clusters. The experimental results are shown in the figures 8.15 to 8.17, the time taken for encoding the files with IKNTN_LZW using BIS is shown in table 8.2.

![Figure 8.14 BIS Implementation of IKNTN_LZW.](image)

Figure 8.14 BIS Implementation of IKNTN_LZW.

![Figure 8.15 Experimental Result of BIS with IKNTN_LZW on sum](image)

Figure 8.15 Experimental Result of BIS with IKNTN_LZW on sum

![Figure 8.16 Experimental Result of BIS with IKNTN_LZW on Xargs.1](image)

Figure 8.16 Experimental Result of BIS with IKNTN_LZW on Xargs.1
Table 8.2 Minimum and Maximum time required to encode bench mark files using Chained Hash Table and BSI implemented IKNTN_LZW Encoding during the experimentation

| S no | File name  | Chained Hash Table MDLZW Encoding time t(|X|) | BIS MDLZW Encoding time t(|X|) |
|------|------------|---------------------------------------------|--------------------------------|
|      |            | Min(t(|X|)) | Max(t(|X|)) | Min(t(|X|)) | Max(t(|X|)) |
| 1    | a.txt      | 483686       | 625593      | 6984       | 16482      |
| 2    | aaa.txt    | 59683936     | 612746782   | 1133092292 | 1178175439 |
| 3    | alice29.txt| 833270336    | 848278783   | 831922772  | 873250930  |
| 4    | alphabet.txt| 535248782    | 554447174   | 494004685  | 512770757  |
| 5    | asyoulik.txt| 688827111    | 704596796   | 681422919  | 699837549  |
| 6    | bib        | 613489935    | 644752682   | 591770742  | 615546669  |
| 7    | bible.txt  | 23614220544  | 23660174346 | 38102368991| 38108093717|
| 8    | book1      | 4463885857   | 4471197856  | 5507170548 | 5651611703 |
| 9    | book2      | 3534541338   | 3583289667  | 4065480913 | 4100331147 |
| 10   | cp.html    | 150022324    | 153177652   | 138947548  | 145242356  |
| 11   | E.coli     | 26201214323  | 26272763223 | 68525369719| 68881544822|
| 12   | Fields.c   | 76307564     | 79537692    | 71185863   | 75413216   |
| 13   | geo        | 602351681    | 618840405   | 569776326  | 586898614  |
| 14   | grammar.lsp| 30706483     | 32973506    | 29092525   | 31359852   |
| 15   | kennedy.xls| 5709508914   | 5892204204  | 5679197202 | 5813136269 |
| 16   | lcet10.txt | 2425988663   | 2529943378  | 2646052780 | 2701500889 |
| 17   | news       | 2193004686   | 2250360016  | 2342728196 | 2438633961 |
| 18   | obj1       | 134902578    | 140205050   | 125831355  | 135876082  |
| 19   | obj2       | 1394457739   | 1413177866  | 1412915050 | 1446662914 |
| 20   | paper1     | 300952663    | 309972756   | 284087503  | 299892305  |
| 21   | paper2     | 456506550    | 468713274   | 439906468  | 457650343  |
| 22   | paper3     | 265885077    | 275793742   | 251411562  | 258136440  |
| 23   | paper4     | 88658745     | 91856970    | 83068099   | 88049458   |
For any type of implementation after each insertion, the corresponding Cluster length is incremented by one, after the final insertion in the dictionary the length of the cluster is represented by \(|C_i| = n_i\) where \(i\) is 1 to \(K\). \(|D| = \sum_{i=1}^{m} n_i\), i.e., the total number of elements in all Clusters is \(C_1\ to \ C_K\).

### 8.4 Computational Complexity Analysis of Various Implementations

**Theorem 8.1** The Linear array with linear search implementation of IKNTN_LZW compression the algorithm takes:

\[
|X| \left( \frac{|D| + K^*2}{K + |D| + 4} \right) \text{ Computation}
\]

**Proof:** The computational cost per cluster is calculated by \(\frac{1}{|C_i|} \sum_{i=1}^{|C_i|} \left( \frac{i+1}{2} \right)\) based on the theorem 5.5 and the average number of comparison required for the input sequence is calculated as follows:

\[
|X| \times \left( \frac{1}{|D|} \left( \frac{1}{|C_1|} \sum_{i=1}^{|C_1|} \left( \frac{i+1}{2} \right) \right) + \left( \frac{1}{|C_2|} \sum_{i=1}^{|C_2|} \left( \frac{i+1}{2} \right) \right) + \left( \frac{1}{|C_3|} \sum_{i=1}^{C_3} \left( \frac{i+1}{2} \right) \right) + \cdots + \right)
\]

\[
\left( \frac{1}{|C_K|} \sum_{i=1}^{C_K} \left( \frac{i+1}{2} \right) \right)
\]

where

\[
\frac{1}{|C_i|} \sum_{i=1}^{|C_i|} \left( \frac{i+1}{2} \right)
\]
\[ = \frac{c_i + 3}{4} \quad (8.3) \]

Then equation 8.1 is simplified as

\[ |X| \left( \frac{1}{|D|} \left( \sum_{i=1}^{K} \frac{|c_i| + 3}{4} \right) \right) \quad (8.4) \]

where

\[ \sum_{i=1}^{K} |c_i| = |D| \quad (8.5) \]

Then the equation 8.4 is simplified as

\[ |X| \left( \frac{1}{|D|} \left( \frac{|D| + (K + 3)}{4} \right) \right) \quad (8.6) \]

\[ = |X| \left( \frac{1}{(K+|D|)} \left( \frac{|D| + (K + 3)}{4} \right) \right) \quad (8.7) \]

By simplifying the 8.7 equation,

where computation required for search in the index table is negligible.

\[ = |X| \left( 1 \cdot \frac{|D| + (K + 3)}{K + |D| + 4} \right) \quad (8.8) \]

\[ = |X| \left( \frac{|D| + (K + 3)}{K + |D| + 4} \right) \quad (8.9) \]

**Theorem 8.2** The Binary search tree implementation of IKNTN_LZW compression algorithm takes:

\[ O \left( |X| \cdot \left( \sum_{i=1}^{K} \log \left( \frac{1}{|c_i|^{1/|c_i|}} \right) \right) \right) \]

**Proof:** The computational cost per cluster is calculated by \( \log \prod_{i=1}^{|c_i|} \frac{1}{|c_i|} \) and the average number of comparison required for the input sequence is calculated as follows:
\[ O |X| \left( \frac{1}{|D|} \left( \frac{1}{K} \left( \log \left( \prod_{i=1}^{|C_1|} \frac{1}{|C_1|} \right) + \log \left( \prod_{i=1}^{|C_2|} \frac{1}{|C_2|} \right) + \log \left( \prod_{i=1}^{|C_3|} \frac{1}{|C_3|} \right) + \right) \right) \right) + \]

\[ \cdots + \log \left( \prod_{i=1}^{|C_k|} \frac{1}{|C_k|} \right) \right) \right) \right) \right) \right) \right) \right) \right) \right)

\[ O |X| \left( \frac{1}{(|D|+K)} \left( \log \left( \prod_{i=1}^{|C_1|} \frac{1}{|C_1|} \right) + \log \left( \prod_{i=1}^{|C_2|} \frac{1}{|C_2|} \right) + \log \left( \prod_{i=1}^{|C_3|} \frac{1}{|C_3|} \right) + \right) \right) \right) \right) \right) \right) \right) \right) \right) \right) \right)

\[ \cdots + \log \left( \prod_{i=1}^{|C_k|} \frac{1}{|C_k|} \right) \right) \right) \right) \right) \right) \right) \right) \right) \right)

\[ O |X| \left( \left( \sum_{i=1}^{|C_1|} \log \left( \prod_{j=1}^{|C_1|} \frac{1}{|C_1|} \right) \right) \right) \right) \right) \right) \right) \right) \right) \right) \right)

\[ \text{where} \]

\[ l(\prod_{j=1}^{|C_1|} j) = |C_1|! \]  \hspace{1cm} (8.14)

Then the equation 8.13 is simplified as

\[ O |X| \left( \left( \sum_{i=1}^{|C_1|} \log \left( \prod_{j=1}^{|C_1|} \frac{1}{|C_1|} \right) \right) \frac{1}{(|D|+K)} \right) \right) \right) \right) \right) \right) \right) \right) \right)

\[ \text{Theorem 8.3} \] The Chained hash table implementation of IKNTN_LZW compression algorithm takes:

\[ O \left( \left( \sum_{j=1}^{|C_1|} \log \left( \prod_{j=1}^{|C_1|} \frac{1}{|C_1|} \right) \right) \frac{1}{(|D|+K)} \right) \right) \right) \right) \right) \right) \right) \right) \right) \right)

\[ \text{Computation} \]

\[ \text{Proof:} \] The computational cost per dictionaries is calculated by \[ \frac{n+\sum_{i=1}^n \alpha_i}{n} \] and the average number of comparison required for the input sequence is calculated as follows:
\[
\frac{1}{|D|} \left( \frac{1}{K} \left( \frac{|C_1| + \sum_{i=1}^{\frac{|C_1|}{|C_1|}} \alpha_i}{|C_1|} \right) + \frac{|C_2| + \sum_{i=1}^{\frac{|C_2|}{|C_2|}} \alpha_i}{|C_2|} + \frac{|C_3| + \sum_{i=1}^{\frac{|C_3|}{|C_3|}} \alpha_i}{|C_3|} + \cdots + \frac{|C_K| + \sum_{i=1}^{\frac{|C_K|}{|C_K|}} \alpha_i}{|C_K|} \right) \right)
\]

(8.16)

By simplifying the equation 8.16 we get

\[
\frac{1}{|D|} \left( \frac{1}{K} \left( \sum_{j=1}^{K} \left( \frac{|C_j| + \sum_{i=1}^{\frac{|C_j|}{|C_j|}} \alpha_i}{|C_j|} \right) \right) \right)
\]

(8.17)

\[
= \frac{1}{(|D| + K)} \left( \sum_{j=1}^{|D|} \left( \frac{|C_j| + \sum_{i=1}^{\frac{|C_j|}{|C_j|}} \alpha_i}{|C_j|} \right) \right)
\]

(8.18)

\[
= \left( \sum_{j=1}^{K} \left( \frac{|C_j| + \sum_{i=1}^{\frac{|C_j|}{|C_j|}} \alpha_i}{|C_j|} \right) \right) \frac{1}{(|D| + K)}
\]

(8.19)

where computation required for search in the index table is negligible.

\[
\frac{|C_1| + \sum_{i=1}^{\frac{|C_1|}{|C_1|}} \alpha_i}{|C_1|} = 1 + AM\{\alpha_j\} \text{ then}
\]

where AM is arithmetic mean,

the equation 8.19 is simplified as

\[
O\left( \sum_{j=1}^{K} (1 + AM\{\alpha_j\}) \right) \frac{1}{(|D| + K)}
\]

(8.20)

**Theorem 8.4** The BSI implementation of MDLZW compression algorithm takes:

\[
O |X| \ast \left( \sum_{i=1}^{K} \log \left( \frac{1}{|C_i|} \right) \right) \frac{1}{(|D| + K)} \text{ computation}
\]

Proof: The computational cost per cluster is calculated by \( \log \left( \prod_{i=1}^{\frac{|C_i|}{|C_i|}} \frac{1}{|C_i|} \right) \) and the average number of comparison required for the input sequence is calculated as based on the theorem 8.2 is,

\[
O |X| \ast \left( \sum_{i=1}^{K} \log \left( \frac{1}{|C_i|} \right) \right) \frac{1}{(|D| + K)}
\]

(8.21)
Figure 8.18 Times taken by Linear array implementation of IKNTN_LZW Encoding algorithm

Figure 8.19 Time taken by BST, Chained Hash Table and BIS Implementation of IKNTN_LZW
8.5 Summary

The major feature of conventional implementations of the LZW data compression algorithms is that they usually use single dictionary only. This takes linear time in encoding. The MDLZW architecture improves and reduces the computational cost of LZW algorithm that has been discussed in Chapter VII. This chapter proposes and explains four novel implementations. The proposed algorithm combines the features of IKNTN clustering algorithm with LZW. This chapter experiments various data structures and algorithms used with the IKNTN_LZW architecture, namely the data structures used for the experimentation, linear array with Linear Search, Binary Search Tree (BST), and Chained Hash table. The experimental results show that the IKNTN_LZW architecture is the best with any data structure and algorithm when compared to the simple LZW and MDLZW architecture. When the results of various implementations of IKNTN_MDLZW, the chained hash table shows the better result when comparing with the other implementations, i.e., IKNTN_LZW shows 81.03% than IKNTN_LZW with Linear array, 20.161% than BST. When comparing with IKNTN_LZW_BIS the IKNTN_LZW with chained hash table achieved 38.642%, BST implementation of IKNTN_LZW computational cost increased 11.930% than BST implementation MDLZW, chained hash table implementation of IKNTN_LZW improved 14.541% than chained hash table implementation MDLZW, BIS implementation of IKNTN_LZW improved 4.85% than BIS array implementation MDLZW and Linear array implementation of IKNTN_LZW improved 91.347% than Linear array implementation of MDLZW.