Hash based Frequent Pattern Mining approach to Text Compression

C. Oswald*, S. Srinidhi†, K. Sri Vishnu†, T.V.Vishalι, B. Sivaselvan*

*Department of Computer Engineering,
Indian Institute of Information Technology, Design and Manufacturing Kancheepuram, Chennai, India.
{coe13d003, sivaselvanb}@iiitdm.ac.in
†Department of Information Technology, Sri Sivasubramaniya Nadar College of Engineering, Chennai, India.
ιDepartment of Computer Science & Engineering, Sri Sivasubramaniya Nadar College of Engineering, Chennai, India.
{srinidhisridharan89, srivishnukumar.k, vishal.vasudevan.vishal}@gmail.com

Abstract. The paper explores the compression perspective of Data Mining. Huffman Encoding is enhanced through Frequent Pattern Mining, a non-trivial phase in Association Rule Mining(ARM) technique, in the field of Data Mining. The seminal Apriori algorithm has been modified in such a way that optimal number of patterns(sequence of characters) are obtained. These patterns are employed in the Encoding process of our algorithm, instead of single character based code assignment approach of Conventional Huffman Encoding. Our approach is built on an efficient hash based data structure, which minimizes the compression time by employing an efficient and novel technique for finding the frequency of the patterns. Simulation over benchmark corpus clearly shows the efficiency of our proposed work in relation to Conventional Huffman Encoding in terms of compression ratio and time.

Keywords: Apriori algorithm, Compression Ratio, Frequent Pattern Mining, Huffman Encoding, Lossless Com-pression

1 Introduction

The Internet, which has changed our lives drastically, involves data in various forms. Mediums that involve huge data transfer rely on compression and decompression approaches to enhance the efficiency of the data transfer process. Data can be of various forms namely text, image, audio, video etc. The real challenge lies in efficiently storing the huge amount of data in a condensed form and reducing their transfer time. The main aim of compression algorithms is to reduce the storage space of large volumes of data and the time taken for their transfer. Compression is of two different types namely lossless and lossy.

The process of reconstructing original data from compressed data is termed as lossless compression. It is gener-ally applied to text documents and source files. Some of the popular techniques are Huffman Encoding, DEFLATE, LZ77, Lempel-Ziv-Welch(LZW), LZ78, Prediction by Partial Matching(PPM), ZIP, Run Length Encoding(RLE), BWT, etc [1]. One of the most commonly used algorithm is Huffman Encoding [2].

Lossy Compression is a class of data encoding method that uses inexact approximations. Such techniques find extensive applications in the transfer, reproduction and storage of multimedia data(image/audio/video) where slight data loss might not be noticed by the end user. It achieves a better compression ratio as compared to lossless compression. MPEG, MP3, JPEG, JBIG, PGF, WMA etc. are a few techniques based on this principle
[1]
. Our work focuses primarily on text compression which is strictly lossless. Huffman Encoding, a seminal algo-rithm for lossless data compression, developed by David A. Huffman, involves assigning variable code length to characters based on their probability of occurences [2]. In this method, every character encoded can be uniquely decoded as the codes generated by Huffman Encoding are prefix codes and this property ensures lossless decod-ing. Compression, Approximation, Induction, Search and Querying are the different perspectives of Data Mining identified by Naren Ramakrishnan et al [3]. In this research, we explore the compression perspective of Data Mining.

The process of data mining focuses on generating a reduced(smaller) set of patterns(knowledge) from the orig-inal database, which can be viewed as a compression technique. FPM, a non trivial phase of ARM is incorporated in Huffman Encoding, which is a lossless compression technique [2,4]. We exploit the principle of assigning shorter codes to frequently occurring patterns(sequence of characters) in relation to single character based approach of Huffman Encoding. Moreover, this work concentrates on employing an efficient data structure to reduce the time to compress the text. The benefits of the Data Mining approach to compression, resulting in an efficient and a novel compression algorithm is indicated by our simulation results. The proposed paper merits applications across domains that employ text data such as data from Web based communication systems like email, facebook, twitter etc., protein sequence database from Bioinformatics domain, source code repositories etc.

Frequent Pattern Mining(FPM) is a non-trivial phase of ARM and is formally defined as follows: Let $I = \{i_1, i_2, i_3, \ldots, i_n\}$ be a set of items, and a *transaction database* $TD = \langle T_1, T_2, T_3, \ldots, T_m \rangle$, where $T_i (i \in [1...m])$ is a transaction containing a set of items in I. The *support* of a *pattern* X, where $X \subseteq I$, is the number of transactions containing X in TD. A Pattern(X) is frequent if it's support satisfies a minimum support($min\ supp = \alpha$) which is user-defined. From Apriori, which is a seminal algorithm in FPM we use the prior knowledge which is, "*All nonempty subsets of a frequent itemset/pattern must also be frequent*" [4, 5]. All possible frequent patterns in the Transaction Database are mined in FPM algorithms. The rest of the paper is organized as follows. Related studies are presented in Section II. A formal presentation of the problem is given in Section III. Proposed algorithms are given in Section IV. In Section V, we present details of the datasets, results and discuss the performance. Conclusion with further research directions are given in Section VI.

2 Related Work

Shannon-Fano coding, a seminal algorithm in text compression, aims at constructing a prefix code which is based on a set of symbols and their occurence probabilities but doesnot achieve optimal codeword length [6]. Huffman Encoding proposed in 1952 is a simple technique which assigns short codes to frequently occuring symbols and it is prefix free [2]. RLE is a simple compression technique which encodes sequences of identical characters occurring together and it is effective only for runs of length ≥ 3 and is more relevant in the domain of images [7]. Arithmetic Encoding developed at IBM is an optimal entropy coding strategy and assigns integer codes to the individual symbols [8]. Adaptive/ Dynamic Huffman Encoding is normally too slow to be practical [9]. It does not generate optimal results on most real data. Several variations of Huffman Encoding exist such as Canonical Huffman Encoding [10], Modified Huffman Encoding [11] etc. PPM is based on an encoder which maintains a statistical model for the text called the order-N encoder [12]. It gives good results based on some assumptions. Start/Stop code, Burrow Wheeler Transform, Self-Deleting Codes, Location Based Encoding, Elias codes, Move to Front Encoding, etc are some of the other techniques [1, 13–15]. These methods exhibit single character or word based encoding mechanism and the way in which the statistical model is built, determines the quality of compression. In Tunstall code, which is a fixed length code with reduced compression size, the demerit is that both encoder and decoder needs to store the complete code [1]. In Golomb Codes, codes are assigned based on the probabilities [16]. For geometrically distributed data items, they are best suited for compression, because they are parameterized prefix codes. But if the value of its parameter 'p' is very large, there is almost no compression.

The family of Lempel-Ziv algorithms comes under the sliding window based method of text compression in which LZ77 is the seminal work [1, 17, 18]. Here, a static/dynamic dictionary is used to encode each string. The variants are LZX, SLH, LZR, LZSS, Statistical Lempel-Ziv, LZB, LZPP, LZMA, ROLZ, LZH, LZHuffman, etc [1, 17]. Variations of LZ77 have an assumption that patterns present in the input text occur closely and this is not true always. In LZ78 class of algorithms, a tree is used to maintain the dictionary where available memory size is limited and they have complex decoding than LZ77 [19]. Other variations are LZT, LZRW1, ZIP, LZP1, DMC, Context Tree Weighting, Win RAR, RAR, LZJ, LZP2, GZIP, bzip2, LZW, LZFG, UNIX Compress, V.42*bis*, CRC etc [1,15,17,20–22]. Limitations such as huge memory requirement, low compression and time inefficiency, relying on word based encoding are present in most of these above techniques. [15] has shown that text compression by frequent pattern mining technique is better than conventional Huffman but time taken to compress is more. Our approach require lesser time than [15] using efficient hash based Frequent Pattern Mining mechanism.

Moving towards FPM algorithms, all frequent itemsets are generated by Apriori and in one scan on the TD, firstly it generates the set of all frequent single length itemsets L_1. Candidate itemsets C_k are iteratively generated from L_{k-1} and those itemsets whose subsets are infrequent, are pruned. Until no candidate itemsets can be generated, this is iterated. The set of all frequent itemsets present in the TD are consolidated at the end [5]. Unlike the level wise approach of Apriori-like algorithms, FP-Growth algorithm is based on a divide and conquer approach [23]. A FP-Tree is generated and recursive mining with that tree is performed. Two scans are enough to generate all the frequent patterns. For very large datasets, the construction of the FP-Tree takes huge space and is tough to accommodate the entire tree in the main memory. For these two algorithms, the transactional database that is represented is horizontal. Counting interface algorithm is an optimization of the Apriori algorithm [24]. Based on equivalence relations between frequent patterns, this algorithm is centered. The support count of one pattern can be determined from an equivalent pattern using these equivalence relations.

When the data is weakly correlated, the performance of this algorithm is very similar to the original Apriori algorithm. Zaki et al proposed a vertical Transactional Database approach called Equivalence Class Transformation [25]. All frequent itemsets by simple Tid list intersections are enumerated in this algorithm. A lattice-theoretic approach is used to partition the data space into smaller pieces which can be processed independently. The search for patterns can be either Top down, Bottom up or Hybrid. H-Mine is an algorithm which exploits a hyperlinked data structure called H-Struct [26]. The performance of this algorithm is high, when the database and its projections are able to fit in the main memory. If the main memory is not sufficient, a partitioning technique is used. The overall performance of this algorithm is better than Apriori based algorithms. In FPM, other algorithms include Dynamic Itemset Counting(DIC) [27], Diffsets [28], Counting Inference [29], Sampling [30], LCM ver.3 [31], RELIM [32], Partitioning [33], Opportunistic Projection, [34] etc. A detailed survey can be seen in [26, 35].

3 Problem Definition

A few notations introduced in this work are given. For a pattern p in T , absolute frequency(f_{abs}) is defined as the number of times p occur as a sequence in T. In this work, a sequence is considered to be the character in an ordered and contiguous manner w.r.t the index. Consider the text '*adracapadra*' where '*ara*' is a sequence in conventional terms but in our work, '*adra*' is only considered as a sequence. The terms sequence and pattern are used interchangeably in the rest of this work. The set of all patterns with their respective f_{abs} is in P . Modified frequency(f_{mod}) of a pattern p' is defined as, $|\{p'|p' \sqsubseteq T\}|$, which denotes the number of times p' occur in an non-overlapping manner in text T (T is stored as an array A). P' contains the set of all patterns with f_{mod} and $|P'| \leq |P|$. Through f_{mod}, the issue of overlapping characters between sequences in T is eliminated.

For an example, given $T =$ *adracapadra* and $\alpha = 2$, the sequence "*a*" from P has $f_{abs} = 5$, but its $f_{mod} = 1$. The f_{mod} of a pattern depends on the f_{mod} of it supersequences in the set P'. The f_{mod} of pattern "*a*" is calculated after considering the f_{mod} of pattern "*adra*" and then deleting "*adra*" from the array A. If the frequent pattern "*adra*" in P'(also a supersequence of "*a*") with $f_{mod} = 2$ is given priority for encoding, *a*'s count has already been considered four times in the pattern "*adra*", which leads to f_{mod} of sequence "*a*" being 1. A formal definition of Frequent Pattern based Huffman Encoding(FPH) problem for a Text is, "Given an input file T of size z, we need to generate the set(P) of all patterns(frequent and candidate) with its respective $f_{abs}(\geq \alpha)$ in T . Classical Huffman Encoding is applied over P', which is constructed from P, to generate the file T' of size z' where $z' << z$."

4 Proposed Hash based Frequent Pattern based Huffman Encoding(FPH) Algorithm

Our algorithm performs efficient compression by assigning shorter codes to frequently occurring patterns which are phrases and longer codes to infrequent patterns. This greatly reduces the size of the encoded text when compared with conventional Huffman Encoding. Our approach prunes patterns which are not used in the encoding method, thereby reducing the size of the code table significantly. The input text is stored as a hash data structure, implemented using separate chaining, which minimizes the time to generate patterns in an efficient way. The size of the Hash Table is 128 ASCII characters, which are arranged in an ascending order. This is done to access the characters in the Hash Table in an efficient way. ASCII values are taken as keys, for the location of 1-length characters. T is scanned once and the indices of the characters are added to their respective separate hash chains. Our FPH strategy is explained in algorithm $\bar{1}$.

The *FP Gen* procedure takes text T and α as input and employs Apriori Algorithm to generate the frequent patterns. We have restricted ourselves with Apriori for use in our proposed technique due to the advantage of its level wise pruning strategy of infrequent patterns. The reason for not choosing FP growth and its successors was, those algorithms require expensive operations for pruning infrequent patterns because of the need to wait for the complete data structure construction which generates the entire set of frequent patterns. Apriori takes the advantage that, frequent patterns are generated as and when the levels are formed. All the unique characters are added to the set L_1. Further candidate patterns are generated from two patterns a, b in L_{k-1} satisfying the condition such that $k-2$ length suffix of a is same as $k-2$ length prefix of b. The new candidate c is generated by concatenating b with the first character of a.

For example, consider the patterns *pla* and *lan* in L_3. To generate a candidate pattern of length $k = 4$, $k-2$ length substrings i.e. $k-2$ length suffix of *pla* and $k-2$ length prefix of *lan* are considered. The respective substrings are *la* and *la*. The new candidate is now obtained by concatenating the strings *p* and *lan*. The pattern *plan* is a generated candidate pattern. The f_{abs}(absolute count) of the candidate pattern c in the original text is counted using the *find count* procedure. For a pattern p, if $p.f_{abs} \geq \alpha$, it is added to the set L_k. All patterns satisfying the condition constitute the set P . A brief illustration with an example is given in Table 1.

The patterns in P are arranged in descending order of its length to give priority to longer patterns. If there is a tie between length, then ascending order of its ASCII value of the characters are considered. *find count* procedure makes use of the hash table to find the f_{abs} of a pattern in T. It takes the pattern and hash table as parameters. The pattern is checked at the index of its first character in T with the help of the hash table. Consider the hash table in Figure

1. For the pattern *plan*, the index of the first character p is read from the hash table. The character p occurs at index 15 and so the pattern is searched from index 15. The count is incremented if the pattern occurs in T starting from index 15.

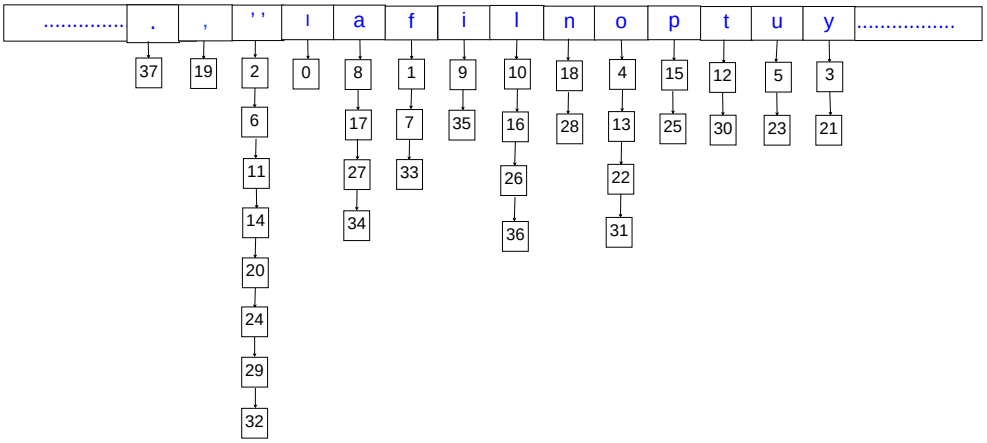

Fig. 1: **Hash Table for the input text** T

Table 1: **Frequent patterns present in** T

Input Text(T): *If you fail to plan, you plan to fail.* $\alpha = \mathbf{2}$.	
Candidate Frequent Patterns(C_k) with its f_{abs}	**Frequent Patterns(L_k) with its f_{abs}**
C_1: (1-length characters) = {.-1 , ,-1 , ' '-8, I-1, a-4, f-3, i-2, l-4, n-2, o-4, p-2, t-2, u-2, y-2}	No pruning in L_1
C_2 = {If-1, f' '-1, ' 'y-2,. . ., pl-2, la-2, ,' '-1,. . ., ' 'f-2, l.-2}	L_2={' 'y-2, yo-2, ou-2, u' '-2, ' 'f-2, fa-2, ai-2, il-2, ' 't-2, to-2, o' '-2, ' 'p-2, pl-2, la-2, an-2}
C_3={' 'yo-2, you-2,. . .,u' 'f-1, ail-2}	L_3 ={ ' 'yo-2, you-2, ou' '-2, ' 'fa-2, fai-2, ail-2, to' '-2, ' 'to-2, ' 'pl-2, pla-2, lan-2}
C_4={' 'you-2, you' '-2,. . ., plan-2}	L_4 ={ ' 'you-2, you' '-2, ' 'fai-2, fail-2, ' 'to' '-2, ' 'pla-2, plan-2}
C_5 = {' 'you' '-2, ' 'fail-2, ' 'plan-2}	L_5 = {' 'you' '-2, ' 'fail-2, ' 'plan-2}
$C_6 = \emptyset$	$L_6 = \emptyset$
P(Arranged in the descending order of pattern length and in ascending order of ASCII values) = $L_{k\geq 2} \cup C_1$ = {' 'fail-2, ' 'plan-2, ' 'you' '-2, ' 'fai-2, ' 'pla-2, ' 'to' '-2, ' 'you -2, fail -2, plan-2, you' '-2, ' 'fa-2, ' 'pl -2, ' 'to -2, ' 'yo -2, ail -2, fai-2, lan -2, ou' ' -2, pla -2, to' ' -2, you -2,' 'f-2, ' 'p -2,' 't -2, ' 'y-2, ai -2, an -2, fa -2, il -2, la -2, o' ' -2, ou -2, pl -2, to -2, u' ' -2, yo -2, ' '-8, ,-1, .-1, I-1, a-4, f-3, i-2, l-4, n-2, o-4, p-2, t-2, u-2, y-2}	

find f_{mod} procedure takes the set of all frequent patterns $P = \cup_k L_k$, T and hash table HT as parameters and generates f_{mod} of the patterns in P. A boolean array $delVal$ of size equal to the size of the input text is used to indicate whether a pattern(including a single character) has been counted previously or not. All the values in the boolean array are initialized to 0. The f_{mod} of each pattern is found by scanning T, with the help of the index of the first character of the pattern which is found using the hash table which is illustrated in Figure 2 and 3. For example, in P, f_{abs} of pattern f was 3. After the f_{mod} of its superpatterns are calculated, f_{mod} of f is found. Since, f has occurred two times in its superpattern $''fail$, f_{mod} of f will be $3 - 2 = 1$.

I	f	' '	y	o	u	' '	f	a	i	l	' '	t	o	' '	p	l	a	n	,	' '	y	o	u	' '	p	l	a	n	' '	t	o	' '	f	a	i	l	.
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fig. 2: Initial Boolean Array $deletedVal$ for $find\_f_{mod}$ procedure

I	f	' '	y	o	u	' '	f	a	i	l	' '	t	o	' '	p	l	a	n	,	' '	y	o	u	' '	p	l	a	n	' '	t	o	' '	f	a	i	l	.
1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Fig. 3: Boolean Array after counting the pattern I in P

The f_{mod} is incremented if the pattern gets a match in T and also the corresponding values in the array $delVal$ representing the characters in the pattern are 0. When a match for a pattern is found, the values in the array $delVal$ corresponding to the characters of the pattern are updated to 1. The patterns satisfying the condition f_{mod}>0 is

Algorithm 1 (FP_Huffman) Encode given text T using frequent patterns.

Input: An Input Text T and α.
Output: Compressed Text T'; *root*: Root of the FP Huffman Tree
Method:
$HT \leftarrow Create\ Hash(T)$ //Creates a Hash Table for the Input Text.
$P \leftarrow FPGen(T, HT, \alpha)$ //Generate Patterns using Modified Apriori
$P' \leftarrow find\_fmod(T, HT, P, \alpha)$ Algorithm.//Generate Modified frequency and adds the patterns to P
$FP\_root \leftarrow FPHuffman\ Tree(P')$ // root of the tree is returned.
$code \leftarrow$ NULL // Code for root is NULL.
$Assign\_code(FP\_root, code, code\_table\langle pattern, code\rangle)$ // Codes are stored in the code table
$T' \leftarrow Encode(codetable, T)$ //T is encoded using the code table.
Return T'

Procedure 1 Create hash Create a Hash Table for T, implemented using Separate Chaining.

Input: An Input Text T. $|T| = n$.
Output: HT: Separate Chaining Hash table for the given text. **Method:**
for (each $c_s(s \in [1, \ldots, n])$ **do**
Insert($HT[c_s]$, s) // Inserts the index s into the corresponding chain of the Hash. **end for**
return HT

added to the set P'. Figure 4 represents FP Huffman tree for T using patterns from P', adopting strategy as like conventional Huffman Encoding. Patterns in code table are stored in the descending order of pattern length and for tie in length, in ascending order of ASCII values. The process of decoding is the same as Huffman decoding.

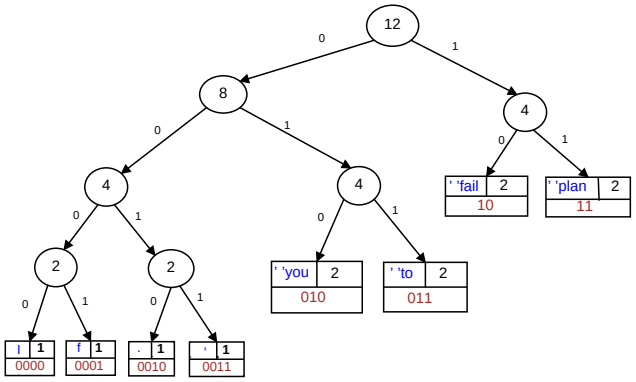

Fig. 4: **FP Huffman Tree for** T

5 Simulation Results and Discussion

Simulation is performed on an Intel core i5-3230M CPU 2.26 GHz with 4GB Main Memory and 500GB Hard disk on Ubuntu 13.04 OS Platform. Using C++ 11 standard, proposed algorithm and Conventional Huffman Encoding are implemented . Canterbury, UCI Machine Learning Repository, TREC and de bruijn corpus as given in Table [36–38] are the standard datasets which the algorithms have been tested over. Table 2 and 3 illustrates the compression ratio of various benchmark corpus that has been tested.

In Figure 5, the efficiency in terms of Compression Ratio(C_r) of the proposed algorithm in relation to Conven-tional Huffman at varying α values for bible and Census-Income dataset is highlighted. The Compression ratio C_r is defined as :

$$C_r = \frac{\text{Uncompressed size of Text}}{\text{Compressed size of Text}}$$

The sum of the code table size(CTS) and the encoded text size(ES) post FPH compression denotes the com-pressed size. As a result of reduced frequent patterns at higher α values, degraded compression is observed from

Procedure 2 FP_Gen Efficient Apriori based mining of frequent patterns.

Input: An Input Text T, Hash Table HT and α.
Output: T: Set of frequent patterns along with their f_{abs}.
Method:
Scan T once.
$C_1 \leftarrow$ Set of all unique characters in T
$L_1 \leftarrow C_1$
$k \leftarrow 2$
$L_2 \leftarrow$ (pattern $c \in C_2$) $\wedge$ $(c.f_{abs} \geq \alpha)$
while $(L_{k-1} \neq \phi)$ **do**
  for (each $a \in L_{k-1}$) **do**
    for (each $b \in L_{k-1}$) **do**
      if ($(k-2)$ length suffix of a = $(k-2)$ length prefix of b) **then**
        $p \leftarrow a[0] + b$ //p is a new pattern generated by concatenating b with the first character of a
        $f_{abs} \leftarrow find\_count(HT, p, T)$ // finds the count of the pattern p.
      end if
      if $(f_{abs} \geq \alpha)$ **then**
        $x.f_{abs} \leftarrow f_{abs}$ //x is the new candidate.
        x.pattern $\leftarrow p$
        $L_k \leftarrow L_k \cup x$ // Add the new candidate x to L_k
      end if
    end for
  end for
end while
return $L = \cup_k L_k$

Procedure 3 : find_count. Finds the f_{abs} of a pattern p in T

Input: An Input Text T, Hash Table HT, Pattern p.
Output: f_{abs} of a pattern p.
Method:
$c \leftarrow p[0]$ // Assign first character of pattern p to c
for each $(ind \in HT[c])$ **do**
  //ind is the value of the index present in the chain $HT[c]$.
  for (j in 0 to $\mid p \mid -1$) **do**
    if $(T[ind + j] = p[j])$ **then**
      $len = len + 1$ // Counting the number of characters that match.
    end if
  end for
  if $(len = \mid p \mid)$ **then**
    $p.f_{abs}$++ // Counting the occurrence of the pattern p if all the characters have matched.
  end if
end for
return $p.f_{abs}$

Table 2: **Details of the various benchmark datasets**

File	Size [Bytes]	Description	Type of data
annhi	17, 154	Medical Records	Alphanumeric data
Alphabet	100, 000	English Alphabets in random order	English text
alice29	1, 52, 089	Novel Collection	English text
bible	4, 047, 392	The King James version of the bible	English text
Census-Income	10, 485, 371	Income Census data of the US	English text
annhiev	21, 834	Genomics biomedical data	Alphanumeric data
de bruijn Sequence	19, 681	Cyclic sequence of a given alphabet	numerical
lcet 10	426, 754	Proceedings on workship on electronic texts	English text

figures 5 to 12. It is observed from figure 5. for bible and Census corpus, the maximum C_r for FPH algorithm is 2.86 and 14.90 at $\alpha = 0.01\%$ which is against Conventional Huffman ratio of 1.82 and 1.75. The proposed algorithm achieves C_r efficiency of 57.14% and 751.68% respectively in relation to Conventional Huffman(CH). C_r efficiency is defined as,

Procedure 4 find_fmod Finds the modified frequency of patterns present in P using the Hash Table HT

Input: An Input Text T, Hash Table HT, P, α.
Output: P': Set of all patterns and their f_{mod}.
Method:
$delVal$: An array of size equal to the size of T, which stores boolean values to indicate patterns that have been counted in T. Initialize all values in $deletedVal$ to 0
for ($p_i(i \in [1, \ldots, |P|])$ **do**
  $temp \leftarrow p$ // For every pattern in P
  $c \leftarrow temp[0]$
  for (ind in $HT[c]$) **do**
    for (j in 0 to | $temp$ | -1) **do**
      if ((T[ind+j] = $temp[j]$) and ($delVal[ind + j] = 0$)) **then**
        $len = len + 1$ // Count the number of characters that match.
      end if
    end for
    if (len = length($temp$)) **then**
      $f_{mod}++$ // Counting the occurrence of the pattern $temp$ if all the characters have matched
      for (m in ind to ind+length($temp$)-1) **do**
        $delVal[m]$=1
      end for
    end if
    if ($f_{mod} > 0$) **then**
      p'_i. $f_{mod} \leftarrow f_{mod}$
      $P' = \cup p'_i$ // Add the pattern to P' if $f_{mod} > 0$
    end if
  end for
end for
return P'

Table 3: C_r **for various benchmark corpus**

File Name	min_supp(%) at max C_r	#Patterns in P	# Patterns in P'	Conventional Huffman C_r	FPH C_r	Time taken for FPH(sec)	C_r Efficiency (%) of FPH
annhi	0.5	345	91	1.66	**3.29**	0.3	98.19
Alphabet	0.64	4056	3	1.67	**416.66**	18.01	24849.70
alice29	0.06	1063	934	1.84	**2.17**	10.07	25.43
bible	0.01	8718	7548	1.82	**2.86**	1123.3	57.14
Census-Income	0.01	193300	1925	1.75	**14.90**	35934	751.68
annhiev	0.2	947	155	1.61	**3.45**	0.36	114.28
de bruijn Sequence	4	30	8	3.59	**8.39**	0.04	133.70
lcet10	0.02	3747	3080	1.67	**2.21**	43.09	32.33

$$C_r \text{ Efficiency}(\%) = \frac{C_r(FPH) - C_r(CH)}{C_r(CH)} \times 100$$

In Figure 6. in alphabet dataset, compression ratio for our approach is 416.66 which is very high as compared to the conventional Huffman compression ratio of 1.67. This is due to the fact that even though the original count of patterns are more, modified number of patterns are very less, due to the dense nature of the dataset where only 4 patterns occur repeatedly. In the range 0.55-0.63%, compression ratio varies minimally, due to the fact that the number of modified patterns satisfying the condition are same. Efficient C_r for alphabet dataset for FPH is obtained for α in [0.55%, 0.65%]. From the discussion above, it is very clear that FPH achieve better C_r than CH. On increasing the $min\ supp$, $|P|$ reduces. For decompression, the Huffman tree is generated using the frequent patterns with their binary codes, which are stored in the code table. Code table size decreases on increasing λ, because of reduction in the cardinality of the frequent pattern set, however the increase in ES of input data nullifies the advantage of code table size reduction. In our approach, P' contains patterns of length greater than 1 with $f_{mod} > 0$ which increases the cardinality of the pattern set which in turn increases CTS, even though ES decreases slightly. Other datasets also exhibit the similar trend. Maximum C_r efficiency is obtained in Alphabet corpus

which is 24849.70% because of the dense presence of frequent patterns. The proposed algorithm achieves efficient compression for support values in [0.01%, 1%].

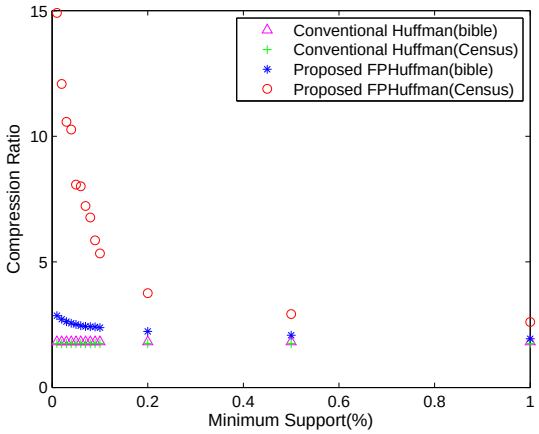

Fig. 5: *min_supp* vs Compression Ratio for Bible and Census

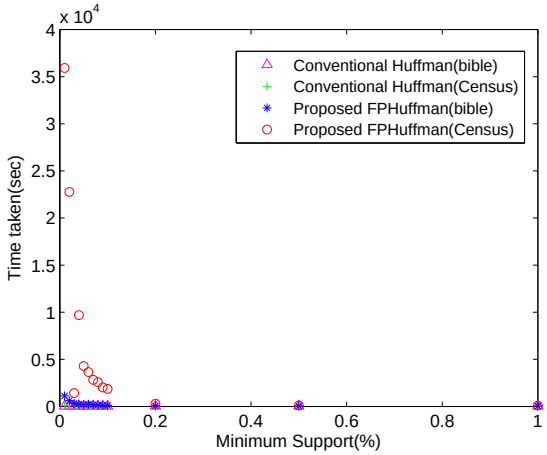

Fig. 6: *min_supp* vs Time for Bible and Census

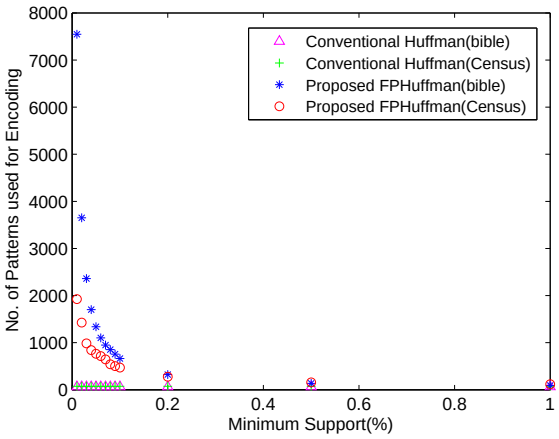

Fig. 7: *min_supp* vs $|P'|$ for Bible and Census

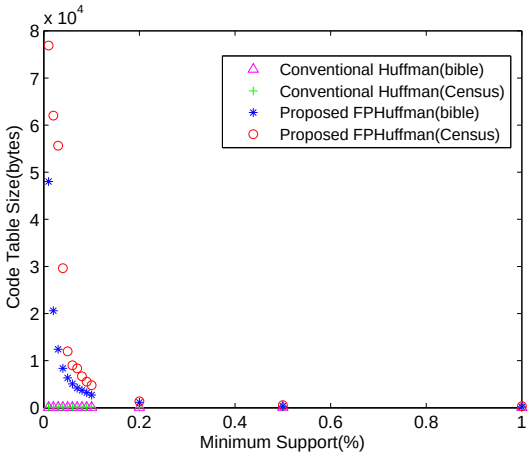

Fig. 8: *min_supp* vs Code table for Bible and Census

The relation between time taken to compress T and C_r is highlighted in figures 6 and 10. In figure 6, for both bible and Census corpus, for our FPH approach in the support [0.01%, 1%], maximum C_r is attained at 0.01%with 1123.3 sec and 35934 sec. as opposed to the CH time of 34.37 and 102.56 sec. In figure 10, for alphabet dataset, time taken by FPH approach is 18.01 as opposed to the time taken by conventional Huffman which is 0.1 sec. For few α values in Census, FPH takes lesser time than CH which is observed from figure 6. For support values in [0.01%, 1%] in Census, even though the cardinality of the pattern set is more than the CH, the encoding time significantly comes down because of capturing lengthier patterns and hence traversing time for T is also less. The encoding time for T in CH encoding increases, because of the need to encode individual symbols. In all datasets except few support values in Census, our proposed algorithm's time is more than the CH strategy because of the number of patterns generated for encoding are more. The pattern generation time in P and in P' increases because of the m scans in T where m is the maximum length of the frequent patterns generated. This time overhead comes from the Apriori algorithm to construct P which is $O(n^3)$. Even though has time overhead, there is a reduction in polynomial time due to non overlapping count between patterns(f_{mod}) which is O n . This polynomial time reduction is achieved using the Hash data structure which helps in efficiently locating the pattern in T . In the worst case, the number of patterns in the set P and in P' can be n. Other tested dataset except alphabet for FPH in α range 0.01 to 1% also exhibit the same trend.

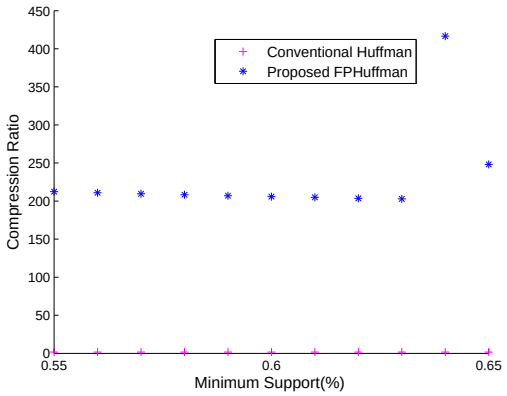

Fig. 9: *min_supp* vs Compression Ratio for Alphabet

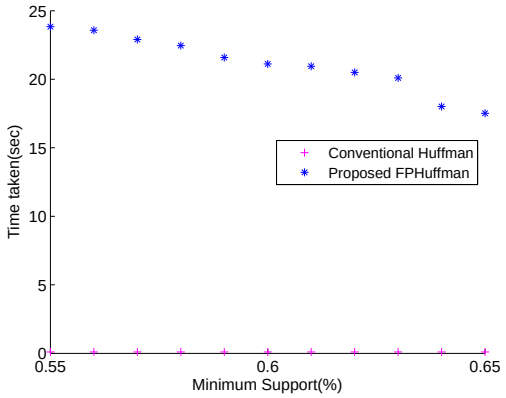

Fig. 10: *min_supp* vs Time for Alphabet

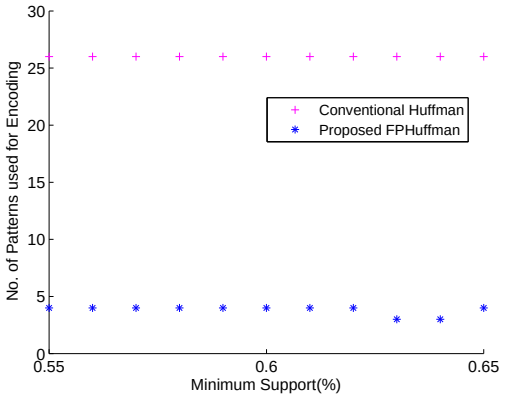

Fig. 11: *min_supp* vs $|P'|$ for Alphabet

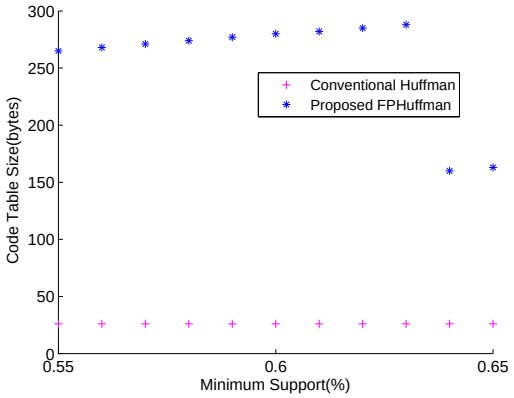

Fig. 12: *min_supp* vs Code table for Alphabet

The relation between α vs $|P'|$ for bible, Census and alphabet corpus is shown in figures 7 and 11. For FPH, for alphabet dataset, for α in [0.55%, 0.65%] and for other datasets for α in [0.01%, 1%], $|P'| << |P|$. The reason is, because of f_{mod}, which avoids overlapping of patterns, which stores only patterns which will be used in the encoding phase. More number of patterns are generated in P because of overlapping between patterns by f_{abs}. In FPH, for the maximum C_r attained at $\alpha = 0.64\%$ for alphabet dataset, $|P'|$ is reduced to 3 from 4056, which is 99.9% reduction. In bible and Census corpus where the maximum C_r achieved at $\alpha = 0.01\%$, the reduction in the pattern base was 13.42% and 99%. Similar observations can be seen in other corpus as well. At some higher support values for few datasets $|P'| = |P|$ or $|P'|$ is not much lesser than $|P|$, because the original count of pattern itself reduces drastically which almost retains the same count in $|P'|$ as well. Moreover, $|P|$ and $|P'|$ converge to the CH strategy since 1-length characters are mostly present. In all the corpus tested, maximum reduction in $|P'|$is obtained at the α value where the maximum C_r is obtained.

The analysis between CTS(in bytes) and α for bible corpus is shown in figure 8. The size occupied by the patterns in P' and its respective codes refers to the code table size. In Figure 8, for bible and Census, for FPH, for α in [0.01%, 1%], CTS is more than its ES and this is because $|P'|$(in FPH) $>>$ $|P'|$ in CH. In figure 12, in alphabet, for α in [0.55%, 0.65%], CTS is more than its ES and this is because $|P'|$(in FPH) $>$ $|P'|$ in CH. At 0.01%, where maximum C_r for bible and Census is achieved, CTS is 48022 and 76886 bytes as opposed to CH Encoding which is 103 and 124 bytes. At $\alpha = 0.64\%$, where maximum C_r for alphabet is achieved, CTS for FPH is 160 bytes whereas CH is 26 bytes. The reason is, the presence of only individual symbols in CH as opposed to sequences of characters of length ≥ 1 which occurs frequently in the code table of FPH. The code table of alphabet is higher than CH, even though $|P'|$ in FPH is only 4, as opposed to CH which is 26. This is because code table also stores the patterns and in FPH, these patterns are much lengthier than the 1-length patterns in CH.

The code table size of our approach for any tested dataset is greater than Conventional Huffman as observed from figures 8 and 12. The encoded size of T for bible, Census and alphabet corpus in our approach are 1363933, 626435 and 80 bytes which is very much lesser than the encoded size of Conventional Huffman strategy which is 2218449, 5989926 and 59615 bytes respectively. In our approach, the CTS is 83.75% more, but the ES is 99.86%

less than conventional Huffman over Alphabet dataset. This contributes to a greater C_r in our approach for α in [0.01%, 0.1%] for bible and Census corpus and alphabet in [0.55%, 0.65%]. The same trend is seen in other corpus as well. This makes FPH approach efficiently than CH approach for all datasets.

6 Conclusion

In this paper, we explored an interdisciplinary and novel text compression algorithm employing FPM in Con-ventional Huffman Encoding strategy. Efficient pattern counting approach was employed to reduce the time to compress and to improve the compression ratio. We show that the proposed FPH algorithm achieves efficient compression ratio, encoded size and execution time. We shall focus on the scope of the FPH algorithm in other lossless word based compression and lossy compression techniques. The compression perspective of Data Mining, Approximation and Search perspectives of Data Mining is also planned to explore.

References

1. David, S.: Data Compression: The Complete Reference. Second edn. (2004)
2. A, H.D.: A method for the construction of minimum redundancy codes. proc. IRE **40**(9) (1952) 1098–1101
3. Ramakrishnan, N., Grama, A.: Data mining: From serendipity to science - guest editors' introduction. IEEE Computer **32**(8) (1999) 34–37
4. Han, J., Kamber, M.: Data Mining: Concepts and Techniques. Morgan Kaufmann (2000)
5. Agarwal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In Bocca, J.B., Jarke, M., Zaniolo, C., eds.: VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile, Morgan Kaufmann (1994) 487–499
6. Shannon, C.E.: A mathematical theory of communication. ACM SIGMOBILE Mobile Computing and Communications Review **5**(1) (2001) 3–55
7. Pountain, D.: Run-length encoding. Byte **12**(6) (1987) 317–319
8. Witten, I.H., Neal, R.M., Cleary, J.G.: Arithmetic coding for data compression. Communications of the ACM **30**(6) (1987) 520–540
9. Vitter, J.S.: Design and analysis of dynamic huffman codes. Journal of the ACM (JACM) **34**(4) (1987) 825–845
10. Sschwartz, E.S., Kallick, B.: Generating a canonical prefix encoding. Communications of the ACM **7**(3) (1964) 166–169
11. Bledsoe, R.E.: Data communication with modified huffman coding (October 1987)
12. Moffat, A.: Implementing the ppm data compression scheme. Communications, IEEE Transactions on **38**(11) (1990) 1917–1921
13. Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm. (1994)
14. Bentley, J.L., Sleator, D.D., Tarjan, R.E., Wei, V.K.: A locally adaptive data compression scheme. Communications of the ACM **29**(4) (1986) 320–330
15. Oswald, C., Ghosh, A.I., Sivaselvan, B.: An efficient text compression algorithm-data mining perspective. In: Mining Intelligence and Knowledge Exploration. Springer (2015) 563–575
16. Golomb, S.: Run-length encodings (corresp.). IEEE Trans. Inf. Theor. **12**(3) (September 2006) 399–401
17. Deorowicz, S.: Universal lossless data compression algorithms. Philosophy Dissertation Thesis, Gliwice (2003)
18. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Transactions on information theory **23**(3) (1977) 337–343
19. Ziv, J., Lempel, A.: Compression of individual sequences via variable-rate coding. Information Theory, IEEE Transactions on **24**(5) (1978) 530–536
20. Nelson, M.R.: Lzw data compression. Dr. Dobb's Journal **14**(10) (1989) 29–36
21. Ramabadran, T.V., Gaitonde, S.S.: A tutorial on crc computations. IEEE Micro (4) (1988) 62–75
22. : Rar implementation. `ttp://www.rarlab.com/rar/unrarsrc-3.5.4.tar.gz`. Accessed: 2016-02-22.
23. Han, J., Pei, J., Yin, Y., Mao, R.: Mining frequent patterns without candidate generation: A frequent-pattern tree approach. Data Min. Knowl. Discov. **8**(1) (2004) 53–87
24. Bastide, Y., Taouil, R., Pasquier, N., Stumme, G., Lakhal, L.: Mining frequent patterns with counting inference. SIGKDD Explor. Newsl. **2**(2) (December 2000) 66–75
25. Goethals, B.: Survey on frequent pattern mining. manuscript (2003)
26. Han, J., Cheng, H., Xin, D., Yan, X.: Frequent pattern mining: current status and future directions. Data Mining and Knowledge Discovery **15**(1) (2007) 55–86
27. Brin, S., Motwani, R., Ullman, J.D., Tsur, S.: Dynamic itemset counting and implication rules for market basket data. In: ACM SIGMOD Record. Volume 26., ACM (1997) 255–264
28. Zaki, M.J., Gouda, K.: Fast vertical mining using diffsets. In: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining, ACM (2003) 326–335
29. Bastide, Y., Taouil, R., Pasquier, N., Stumme, G., Lakhal, L.: Mining frequent patterns with counting inference. ACM SIGKDD Explorations Newsletter **2**(2) (2000) 66–75
30. Toivonen, H.: Sampling large databases for association rules. In: VLDB. Volume 96. (1996) 134–145

31. Uno, T., Kiyomi, M., Arimura, H.: Lcm ver. 3: Collaboration of array, bitmap and prefix tree for frequent itemset mining. In: Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations, ACM (2005) 77–86
32. Borgelt, C.: Keeping things simple: Finding frequent item sets by recursive elimination. In: Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations, ACM (2005) 66–70
33. Savasere, A., Omicinski, E.R., Navathe, S.B.: An efficient algorithm for mining association rules in large databases. (1995)
34. Liu, J., Pan, Y., Wang, K., Han, J.: Mining frequent item sets by opportunistic projection. In: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining, ACM (2002) 229–238
35. Borgelt, C.: Frequent item set mining. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery **2**(6) (2012) 437–456
36. Calgary compression corpus datasets. `corpus.canterbury.ac.nz/descriptions/` Accessed: 2015-07-23.
37. UCI machine learning repository. `https://archive.ics.uci.edu/ml/datasets/Census+Income` Accessed: 2015-10-05.
38. Trec genomics track data. `http://skynet.ohsu.edu/trec-gen/data/2004/` Accessed: 2015-10-05.