International Journal of Engineering and Advanced Technology (IJEAT)
ISSN: 2249-8958 (Online), Volume-9 Issue-1, October, 2019

Novel Space Efficient Indices for Kannada Text: V-KTPY Trie Family

Yashaswini Hegde. Padma S.K

Abstract: ***V-KTPY Trie family is a group of space efficient indices designed to store and search Kannada text. The existing text searching and document fetching methods use different kinds of indices such as indices based on hashing, lexicographical indices and clustering technique based indexing. Each of these indexing methods have their own advantages including the optimal time complexity. However these indices are not space efficient. In this paper we are proposing a family of novel space efficient indices called V-KTPY Tries, which have the features of both lexicographical and hash based indexing. V-KTPY Tries are congruence of V-KTPY Rule ("Vistruta Katapayadi sutra") and Prefix trees (Trie) , where the text labels of the Trie are encrypted by V-KTPY Rule. This powerful rule is an extension of an ancient "Katapayadi Sutra" (KTPY Rule) which can convert characters of Brahmi/Devanagari scripts to numbers. In this paper V-KTPY Tries are indexing V-KTPY encrypted Kannada text due to which compression is possible. The experiments are conducted on the family of V-KTPY Tries and their corresponding Tries with unicode Kannada. And the results show that the simple V-KTPY Trie gives 35% space efficiency; V-KTPY 10Ary Trie gives 65% space efficiency over simple unicode Trie with almost the same time complexity. The Prefix Hashed Trie is a fully compressed V-KTPY Trie which gives 20% space efficiency when compared to fully compressed unicode Trie.***

V-KTPY Tries can be used where Tries are applicable. The V-KTPY prefix hashed Tries are used in Kannada feature selection. V-KTPY Tries can be extended to index many (120+) Indian languages which follow Brahmi or Devanagari script.

Keywords— Indices; VKTPY Tries; Prefix Hashed Trie; Kannada; ಕನ್ನಡ

I. INTRODUCTION

Kannada being a spoken language of around 40 to 60 million people from Karnataka, a southern state of India, generates billions of web pages/documents in Kannada. The current retrieval engines works with unicode versions of algorithms and structures for non-English Languages including Kannada. These existing unicode based techniques are good and capable of handling all the languages of the world. However it is not explored from the perspective of common and similar scripts used in Indian Languages such as Brahmi or Devanagari Scripts. [1]. More than 120 Indian Languages [2] share either Brahmi or Devanagari scripts. In these both scripts the ordering of the alphabets are similar, enabling a possibility of a common numerical representation to all these Languages.

Revised Manuscript Received on October 30, 2019.
* Correspondence Author

Yashaswini Hegde*, Graduate in Electronics and Communications Engineering from UBDT College of Engg, Davangere.
Dr. S.K Padma, Professor, Department of Information Science & Engineering, SJCE College Mysuru.

It is interesting to note that this common numerical representation can have several advantages varied from space efficient data structures to common Computational Phonetic Model(CPM) for Indian Languages. This possibility motivated us to figure out a novel encryption (encoding and decoding) method which would give common numerical representation to as many Indian Languages as possible. We extended Panini's "Katapayadi Sankhya Sutra" (ಕಟಪಯಾದಿ ಸಂಖ್ಯಾ ಸೂತ್ರ) [3] (KTPY Rule [4]) which was used to convert only consonant characters of Devanagari script to numbers and called it as "Vistruta Katapayadi Sutra" (ವಿಸ್ತೃತ ಕಟಪಯಾದಿ ಸಂಖ್ಯಾ ಸೂತ್ರ), (V-KTPY Rule) [5]. This rule is capable of covering many (around 120 [2]) Indian and south east Asian languages and capable of encrypting every alphabet classification such as vowels, consonants, conjugate consonants in an unique way [5].

Table- I: Vistruta Katapayaadi Sankhya Sutra (V-KTPY Rule) ವಿಸ್ತೃತ ಕಟಪಯಾದಿ ಸಂಖ್ಯಾ ಸೂತ್ರ [5]

Grp Name	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*0*
Ka-grp *1*	*k* ಕ	*kh* ಖ	*g* ಗ	*gh* ಘ	*ng* ಙ	*c* ಚ	*ch* ಛ	*j* ಜ	*jh* ಝ	*ny* ಞ
Ta-grp *2*	*T* ಟ	*Th* ಠ	*D* ಡ	*Dh* ಢ	*N* ಣ	*t* ತ	*th* ಥ	*d* ದ	*dh* ಧ	*n* ನ
Pa-grp *3*	*p* ಪ	*Ph* ಫ	*b* ಬ	*bh* ಭ	*m* ಮ					
Ya-grp *4*	*y* ಯ	*r* ರ	*l* ಲ	*v* ವ	*sh* ಶ	*Sh* ಷ	*s* ಸ	*h* ಹ	*L* ಳ	
Swara0 *5*	*a* ಅ	*Aa* ಆ	*i* ಇ	*ii* ಈ	*u* ಉ	*uu* ಊ	*R* ಋ	*Ru* ೠ		

Swara1 6	*e* ಎ	*ee* ಏ	*ai* ಐ	*o* ಒ	*O* ಓ	*ou* ಔ	*am* ಅಂ	*ah* ಅಃ		
Gunita1 7	◌ಾ	◌ಿ	◌ೀ	◌ು	◌ೂ	◌ೃ	◌ೄ			
Gunita2 8	◌ೆ	◌ೇ	◌ೈ	◌ೊ	◌ೋ	◌ೌ	◌ಂ	◌ಃ		

Some more features of the V-KTPY rule listed in [5] are , it

- extends KTPY Rule to numerically represent all characters (Akshara) of Brahmi and Devanagari scripts including Kannada and other Indian languages.
- considers all alphabet classifications mentioned by a great grammarian of ancient India called Panini. These natural classifications based on origin of sounds (phonemes) such as 'swara' (vowels), 'kaaguNita' (Conjunctive consonants), 'ottakshara' (compound consonants) , 'vargeeya vyanjana' (classified consonants), 'avargeeya vyanjana' (miscellaneous consonants), 'yogavaahaka' are given bin (1-8) and index (1-9 and 0) numbers. Only 'pa-grp' is indexed from 1 to 5.
- Table I shows how unique numerical representations are given to all characters. [5]
- Easy to decode back to unicode.

By this V-KTPY rule as given in Table I , each Kannada character can be represented by a two digit number. The first digit indicates the group to which the character belongs to, (nothing but a 'bin number')
and second digit gives the position of the character in that group (bin) as its index. For example the word 'Kannada'(ಕನ್ನಡ) becomes 1120882023 , ದಮಿತ (damita) becomes 28357226 and ಕ್ಷಣಿಕ (kShaNika) becomes 118746257211.

Table-II. Unicode V/s V-KTPY Numerical Representation

	V-KTPY	*Unicode*
1.	Capable giving numerical representations up to 100 alphabets/characters	Capable giving numerical representations up to 127 alphabets/characters for a language
2.	Each alphabet is given a 2 digit number. Ex: 'ಕ ' (ka) = 11	Each alphabet is given a 4 digit number. Ex: 'ಕ ' (ka) = 0C95
3.	Words with Compound consonants like Kannada (ಕನ್ನಡ) = 1120882023	For the same word Kannada (ಕನ್ನಡ) = 0C950CA80CCD0CA8 0CA1
4.	Alphabets used in day to day life to speak and write – are considered.	Considers all regular alphabets and those used in ancient scripts and emojis .
5.	• Simple and compact in terms memory usage . • Same numerical representation to Indian Languages following Brahmi/Devanagari script.	• A multi byte character representation where its size varies from 1 to 6 bytes. [5] • Takes more space as it is capable of representing all global languages.

It is notable that these representations are more compact compared to unicode as shown in Table II . The Table II compares advantages of the V-KTPY encryption over Unicode style encryption. However we don't see our V-KTPY rule as an alternate to unicode since this rule can not encode ancient scripts and emojis and also limited to only Indian Languages. But V-KTPY encryption can be easily decoded back to unicode hence can be used as potential encryption technique .

V-KTPY encrypted text as they are numeric in nature can be used in many data structures and algorithms of Document Retrieval engine. To name a few advantages of such usage are

1. Compact memory usage (lossless compression)
2. Gain in accuracy and precision
3. Defining new features of the language model

To demonstrate the first one of these claims, we have implemented the V-KTPY Rule, computed the memory occupied by the V-KTPY representation of Kannada and compared this with the memory occupied by the unicode representation of Kannada. We have also implemented the simple and compressed V-KTPY Tries, and compared its performances with the unicode Kannada Tries to show that V-KTPY text improves memory usage when stored in data structures like Trie.

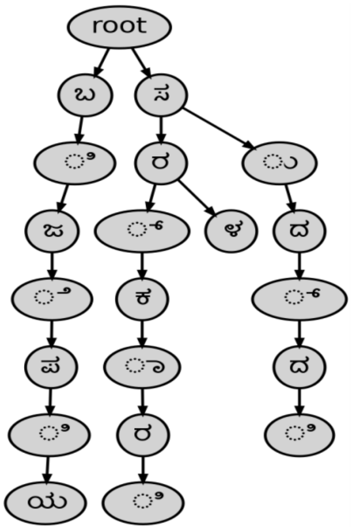

Fig. 1.Simple Kannada unicode Trie [5]

A.Tries

Tries are tree based lexicographical indexing structures. They are widely used to search variable length strings that are stored as text labels in the Trie data structure. Its optimal time complexity is O(dm), where d is the size of a string and m is the size of a alphabet. Hence they are popular as fast data structure. However Tries are space intensive with space complexity O(n) where n is the total size of strings. They are also known as Prefix trees or Radix trees. The text labels stored in the nodes are keys and common prefixes of those text labels are shared by all the descendants of the nodes. The frequencies of the prefixes are stored along with the leaf node details of the Trie.

The Fig. 1. shows the visuals of the standard simple Trie for unicode Kannada words. In simple unicode Kannada Trie each node is labeled with a Kannada character except the root. A Path from the root to the leaves gives a word.
Radix and Patricia trees are the compact Tries. They are space optimized variants of Trie. The compact Prefix Tries or fully compressed Trie or the Radix Trie are obtained by collapsing the single leaf nodes. In other words the only child will be merged with its parent. In the Radix tree the number of children of every internal node is at most the radix r, where $r > 0$ and a power x of 2, where $x \geq 1$ and unlike a simple Trie the edges are labeled instead of nodes.
Unlike other trees where a full key is compared in Radix Trie, the key at each node is compared bits by bits. The multiplies of bits at the node is the radix r of a tree. The Patricia Tree is a special case of the Radix Tree where r=2. In the Patricia tree each bit of a key is compared leading to a two-way split. When $r \geq 2^{\wedge 4}$ it is called an r-ary Trie. As the radix value increases, less is the depth but at the expense of sparsity.
Tries are used for

- Its optimal insert, delete and search time hence for pre processing the patterns to speed up pattern matching.
- Its capability to retrieve all possible complete values which matches a prefix.
- Prefix search with a count. ie To get the the count of words having the common prefix.
- Capability to identify the root word.
- Indexing the sorted text with best time complexity for inserting and searching the patterns.

These applications of Trie inspired us to explore different Tries labeled with V-KTPY encrypted Kannada text.
We explain related works in section II; V-KTPY family of Tries in section III; details of the text corpus used, experiments and results in section IV; conclusion and future work in section V.

II. RELATED WORKS

The Katapayadi sankhya sutra (KTPY Rule) though used in ancient time (1 st CE), it is being considered by the computer scientists recently for its powerful encoding capabilities. This rule is compared with modern hashing technique by Anand Raman from Massey University, New zealand [6], where till today the credit for inventing hashing method was given to H.P. Luhn of IBM (1953). While working in Bhandarkar Oriental Research Institute Subhash kak figures out the possibility of using KTPY rules as binary numbers [4]. Further in his book ”Computation in Ancient India” T.R.N Rao and Subhash Kak, significance of KTPY rule in science of computing , Indian logic and grammar [7].
Trie [8] that has been successfully used in information retrieval . The end node called leaf node which is end of the chain of alphabet labels of a Trie, represents a string [9]. Tries are pretty fast with good insert, delete and search time with good optimal time complexity. Hence are used in huge text management applications. With their good performance k-d digital Trie [10] are used in pattern matching and double array listing [11] . MSD Radix sorting and searching [12] techniques are used in dictionary and text processing. The hash-trees are used in text data mining and compression[13]. Bell et al. in 1990 [13] shows that the space can be saved by reducing the number of Trie nodes. They achieve reduction in memory usage by omitting chains with a single leaf.They call it a compact Trie [13]. Sedgewick, in 1998 [14] proposed a Trie called Patricia Trie with which the compression is further achieved by omitting not only those nodes with single leaves but also an entire chains without branches. The ternary search tree [12] by using a 3-way trie-nodes can reduce the memory usage for sparse data comparisons with greater than, less than and equal to. The Trie literature observes Trie compression [15] , Trie compaction and heuristics [16] . However these techniques, trade off insertion and the search time with space. At first these issues are addressed by Acharya et al.[17]. They developed cache efficient algorithms that choose between several representations of Trie nodes. In 2002, Heinz et al. [18] reduced the number of Trie nodes at little cost. They successfully achieved this by collapsing trie-chains with common prefixes into the same buckets. They named it as Burst-trie . This trie has move-to-front on access strategy while buckets represented as linked lists. [9]. They are then selectively burst into smaller buckets that are parented by a new Trie.
In the context of Kannada language, Tries have been used to figure out the root words by eliminating its prefixes.They are used as the indexing technique. In 2013, Sumant Kulkarni and Srinath Srinivasa worked on Trie indexing and they called this Trie as TrieIR [19].
Though there are several attempts to improve the memory usage in Trie, not many have achieved the average good time complexity and good space complexity in Trie for unicode Kannada Text. In our approach to compress the Trie we have not only adopted previous methods like collapsing the nodes but also a new method of collapsing the nodes by combining the codes for compound and conjunctive consonants. We explain our approach in the following section.

III. V-KTPY TRIE FAMILY

The family of V-KTPY Trie consists of simple Trie , PAT tree and fully compressed Prefix hashed Trie. These Tries are labeled with V-KTPY encrypted Kannada text instead of unicode Kannada text. And hence they are congruence of Tries and V-KTPY encrypting rule. Fig. 1 is a simple Trie constructed for unicode Kannada words and its equivalent V-KTPY Trie is shown in Fig. 2.
We are proposing a new way of node compression method by combining consonant conjunctives with their respective consonants. In Kannada language usually such a combination, for example ‘ಯಾ’ (combination of ‘ಯ ‘ and ‘ಾ’) is treated as a single letter. Such a compression of combination of letters is shown in The Fig. 3. The Fig. 4. shows its equivalent in Kannada unicode. Comparison of these two figures leads to a conclusion that the number of pointers can be reduced in a Trie by using our this simple compression method which improves usage of main memory. This is possible in unicode Kannada also but at the cost search time to search such combinations. Hence we argue that our approach is a better option.

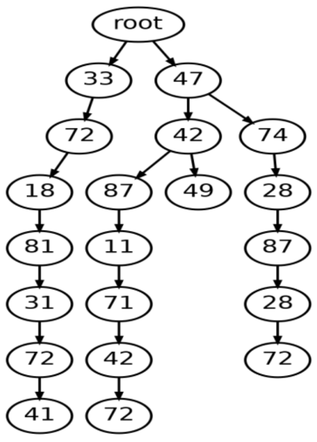

Fig. 2.Equivalent V-KTPY Trie [5]

A. Compressed Simple V-KTPY Trie

A compressed Trie is the one having fewer branches. The elements of the subtrie are not partitioned into more groups in a compressed Trie due to removing all branch nodes having a single child. This feature can improve performance metrics of both time and space.

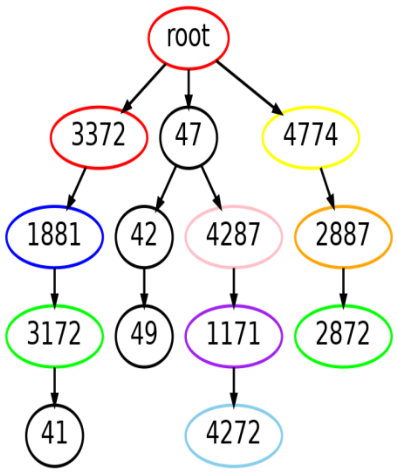

Fig. 3.Compressed V-KTPY Trie [5]

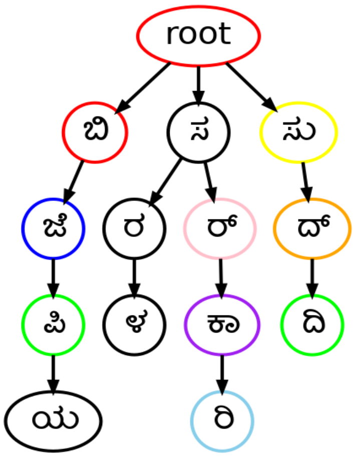

Fig. 4.Unicode equivalent Trie of Fig 3. [5]

1) V-KTPY-Patricia Trie: VKPT

Our proposed V-KTPY-Patricia Trie (VKPT) is very similar to Patricia Trie. Since V-KTPY Trie encodes a Kannada character with each two digits it can easily be compressed like a Patricia Trie with 8-bits key fragments. The Fig. 5 shows the insertion of strings in a VKPT.

The Insertion in VKPT issimilar to Patricia Trie. But in the case of VKPT unlike Patricia with Kannada unicode ,it is enough to compare 8 bits where as in the unicode bits vary from 8 bits to 48 bits. For example if set of strings

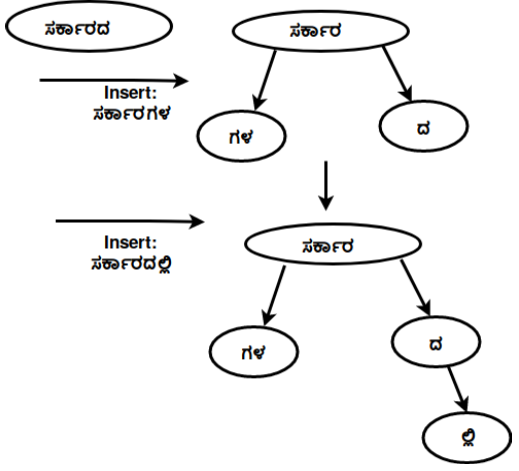

Fig. 5.Insertions: V-KTPY Patricia Trie

S= {ಸರ್ಕಾರದ, ಸರ್ಕಾರಗಳ, ಸರ್ಕಾರದಲ್ಲಿ, ಸರ್ಕಾರವನ್ನು}

its equivalent V-KTPY codes are

V-KTPY = {47428711714228, 4742871171421349, 4742871171422843874372, 4742871171424420872074}.

The last 2 digit of 47428711714228,

4 2 2 8 = 0100 0010 0010 1000 is compared with

4 2 1 3 = 0100 0010 0001 0111

differs after 11th digits and hence it is branched to the left. And in case of 4 2 2 8 = 0100 0010 0010 1000 and 42 13=01000010 00010111 V-KTPY allows fixed and less number of comparisons.

The main advantage of using the V-KTPY with Patricia is even in the case of variable length key, that the number of bits comparison at the max is 8 bits as shown in the above example. Further the height of the tree is reduced since each character is represented by 8 bits only unlike 8 to 48 bits in case of Kannada unicode.

2) V-KTPY-10Ary Trie

Our proposed V-KTPY-10Ary Trie is a higher order compressed Trie just like a Social Security Trie. It is a compressed Trie with digit numbers with variable length keys. Each node of V-KTPY-10Ary Trie has 10 + 1 fields and indices from 0 to 9 and one more for '$'. Along with these fields each node structure has two additional fields.
The branchChar field tells us which digit of the key is used to branch at this node. And fptr field gives the pointer to the first child in a node. Since by V-KTPY rule, a character is represented by 2 digits, the search key is scanned for 2 digits at a time. Hence if the branchChar field has odd number then the following character is from the same bin.
If the value is even number then it means the following character belongs to the same or different bin. Fig. 6. gives the node structure of the V-KTPY-10Ary Trie.

Only the branchChar field is shown in this Figure.

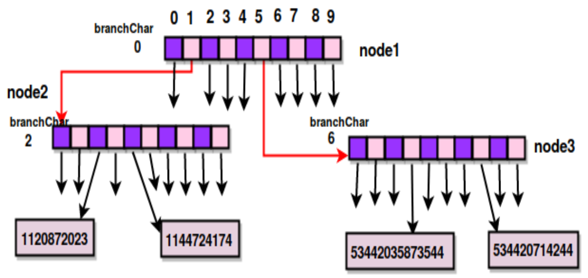

Fig. 6. 10-way V-KTPY Trie Node

A V-KTPY-10Ary trie is searched by following a path from the root. The digit in the branchChar field, at the each branch node is used to determine which subtrie to move to. For example while searching for the key 1120872023 in Fig. 6., we begin with the root. Since branchChar field value is branchChar = 0, the first digit (key positions enumerated from 0 towards right) of the search key is used to choose a branch. In this case it is child[1] and the search pointer is moved to child[1] which is node2. Now since branchChar field value of child[2] is 2, the 2nd digit of the search key is used to branch to the next level. Since child[2] is a leaf field value we get the search key found after a successful comparison. The search key 1120872067 also eventually falls in the same path but search fails due to unsuccessful match. Since V-KTPY-10Ary trie can has variable length keys the prefixes, for example key 11208720, are branched with an ending character '$' as shown in Fig. 7. as case2 with in the box. The search algorithm is explained in Algorithm 1.

Algorithm 1: searchTrie , search in variable length
V-KTPY 10Ary Trie

```
Input : ROOT , K ← V-KTPY key
Initialize: Set P ← ROOT ,N ← currNode
    1. while Until N is NOT a leafNode containing the
       key K do
        1. Get branch char from current node - bchar
           ← N.bchar
        2. Compute Key Index -
           keyIndex ← getIndex(K,bchar)
        3. if keyIndex is found among N.children then
               Get the child with that key index and set it
               as current node
            1. N ← child
        4. else
               Return the node with 0 indicating
               unsuccessful search
            1. return P,0
        5. end
    2. end
       return parent of leafNode, N and non zero keyIndex
       indicating successful search
    3. return N , keyIndex
```

To insert an element with key 538828358835 into the Trie of Fig. 6., we first search for an element with this key.

The search ends at node3 of Fig. 6. Since, the search key and the node key, 53442035873544, do not match, we conclude that the Trie has no element whose key matches with the search key. To insert the this new key, we find the first digit where the search key differs from the key in the node node3, and create a branch node for this digit, a new branchChar.

Since, the first digit where the search key 538828358835 and the element key 53442035873544 differ in the second digit (indexing begins with zero), we create a new intermediate branch node n3, with branchChar = 2 as shown in Fig. 7. Since, the value of branchChar increases as we go down the Trie, the proper place to insert the new branch node can be determined by retracing the path from the root to node3 of Fig. 6. and stopping as soon as either a node with digit value greater than 2 or the node3 is reached. In the Trie of Fig. 6., this path retracing stops at node3. The new branch node, n3 is made the parent of the node n4 (which is node3 in Fig. 6.), and we get the Trie of Fig. 7. case1.

There is another case of insertion as in with the key 53442026873544 into the compressed Trie of Fig. 6. Insertion of this key begins with the search for a leaf node and eventually terminates with the pointer node3.child[6] = null. So to complete the insertion, the leaf node key is sought in the subtrie rooted at node node3. This key is found by traversing through a path from node node3 using the first non null link in each branch node encountered. Thus compressed Trie of Fig. 6., leads us to first leaf of node3 when this procedure is followed.

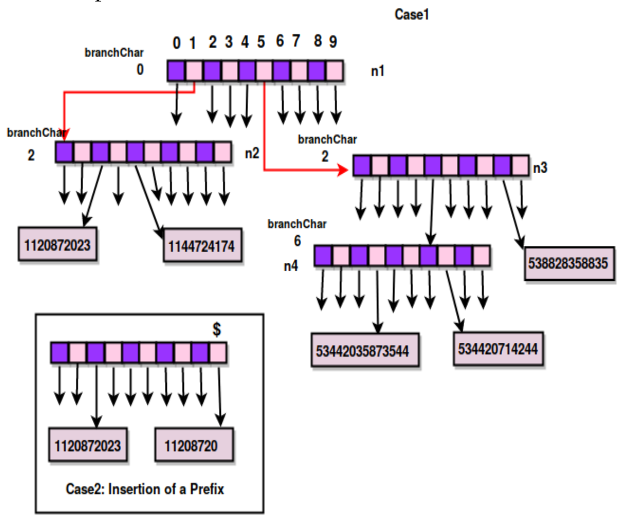

Fig. 7. Insertion: 10-way V-KTPY Trie

After reaching a leaf node of node3, we find the first digit where the leaf node key and the search key differ and complete the insertion as in the previous example. The insertion algorithm is explained in Algorithm 2.

Algorithm 2: insertTrie , variable length V-KTPY key sinsertion in V-KTPY 10Ary Trie

1. Initialize: Set P <- ROOT ,N <- currNode , K is V-KTPY key
2. insertTrie() invokes searchTrie() and K is searched in the Trie, if K is not found in the Trie a new node with K is inserted into the Trie in the appropriate place, and returns an updated V-KTPY populated Trie. If k is found then frequency count of the K is incremented.
3. N, keyIndex <- searchTrie(P,K); The return enumerates three cases

- case1. N is root when Trie empty
- case2. N is node where search stops not finding the key with keyIndex
- case3. N is parent node of leaf key node with keyIndex if Key exists in Trie

```
4. if N.fptr is None: case1 then
  • Creating the first child
  • Get index as first character of key K
  •  Set N.bchar <- None
  • Create a new leaf childlea f containing N as its parent,
    index as the keyIndex and K the key
  • Set firstPointer N. f ptr <- index , a pointer to any leaf in
    the N to index of the key
  • return P
5. end
6. if searchTrie() returns with keyIndex None :case2 then
  • Get child <- f irstLea f , any leaf child in the N node
    pointed by N.fptr
7. else if keyIndex in N.children :case3 then
  • child ← node.children[keyIndex] this step enables 2 cases
c1: child exists with keyIndex and key equal to K
c2: child exists but K is a new a key to be inserted
  • if  child.key is equal to K :c1
then increment K prefix count
return P
end

8. if leaf child is not Null :c2 then
  • Getting the existing leaf details child.K
  • Get new branch char bchar
  • Recompute the index of the child leaf and K due to new
    bcharcldIndex←getIndex(childKey,bchar)
  • ktpykeyIndex ←  getIndex(K,bchar)
  • Get index returns index of the key and if the key is a
    prefix of the existing key in Trie it returns $ thus capable
    of handling the variable length strings
  • Recomputed branch char enumerates three cases
  • cc1. if node branch char none or recomputed bchar and
    currentNode N.bchar are same – indicating insertion in the
    existing node
  • cc2. if recomputed bchar is less than currentNode N.bchar
    indicating new branch created somewhere between root
    and leafparent node.
  • cc3. if bchar is less than root N.bchar then newbranch
    node created above the existing root node and set as root
    of the Trie.
  • cc4. if bchar is greater than root N.bchar then newbranch
    node created at the next level of the Trie.
  • if N.bchar is None or bchar is equal to N.bchar:cc1
then new Node
1. NN←CreateNewLeaf(N, ktpykeyIndex,K)
2. update its branch char and set its fPtr NN.fptr to min of
   cldIndex and ktpykeyIndex
  • else if bchar less than node.bchar then
3. trace the node tracedNode in the Trie from Root upto the
   leaf child node for right place to insert K
4. Create a new BranchNode newBranchNode there
5. if tracedNode.parent is None indicating that it is root: cc2
   then
  • Set tracedNode as a child of newBranchNode with
    cldIndex newBranchNode as the new Root , by its parent
    None
  • Set newBranchNode as parent of tracedNode.parent
  • update branch char and fptr of newBranchnode
  • Set P <- newBranchNode
6. else :cc3 case
  • Set tracedParent as parent of tracedNode
  • Make newBranchNode as a child of tracedParent with
    tracedIndex
  • Set tracedNode as a child of newBranchNode with
    cldIndex
  • Set newBranchNode as parent of tracedNode
  • update branch char and fptr of newBranchnode
  • else :cc4 case
  • newBranchNode with lea f Index is created with bchar
    and K
  • Set child as child of newBranchNode
  • Set newBranchNode as parent of the child
  • update branch char and fptr of newBranchnode
  • Set N as parent of newBranchNode and make
    newBranchNode as N's child
      • return P
```

As suggested by Knuth[29] a generic equation for M-Ary Tries are considered with N numbers of key words for an analysis of these algorithms.

According to Knuth , in a in M-ary Trie , the number of nodes needed to store N random keys with branching terminated for s keys is approximately

$$N/(s \ln M) \qquad (1)$$

This equation 1 is valid for large N small s and small M and for a Trie with M link fields. The “(1)” can be further simplified to

$$N/(\ln M) \qquad (2)$$

if s=M. Thus for around N = 4000 random keys and M = 11 (0 to 9 and one for '$') we require 1668 number of link fields. In our case for 3791 words we get 1568 link fields (excluding leaf links), true to the equation N / ln M.
Further, in V-KTPY-10Ary Trie if the key contains say 9 digits then the height of Trie is <= 10. And search takes nine branches and a single comparison. If the same keys of 9 digits are stored in Red-back Tree then height is around 60 ($2 \log_2 10^9$) and it takes up to 60 memory access and 60 comparisons. It is around 40 in case of AVL Tree and 30 in case of binary tree.

3) VKTPY Prefix Hashed Trie : VKTPY PHT

VKTPY PHT is a special kind of Trie, a compressed one with less number of branches. It is a lexicographical index as well as a hash based index structure.

It is compressed by

1. combining consonant conjunctives
2. storing indexed prefixes, by its first character
3. collapsing all single nodes

In VKTPY PHT, each node is an array structure which stores the prefix and a child pointer. The VKTPY PHT data structure as shown in Fig. 8 has these following features.

- VKTPY PHT root node is similar to a hash table.
- Each bin of the the has table holds a hash key and a pointer.
- This hash key is nothing but V-KTPY encoded Kannada word.
- And the pointer is to point a NAry Tries of the subsequent levels.

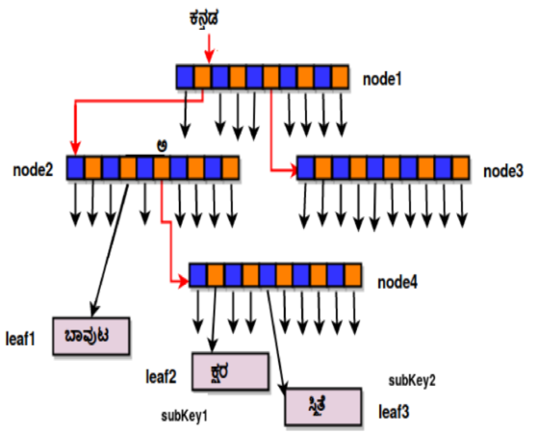

Fig. 8. VKTPY-PHT Node Structure [5]

V-KTPY words with similar prefixes, for example “ಕನ್ನಡಬಾವುಟ"(kannaDabaavuTa) and "ಕನ್ನಡಅಕ್ಷರ" (kannadDaakShara) both have same prefix “ಕನ್ನಡ" (kannaDa) are hashed to same location in Nary as they have same numeric representation “ 1120882023” . Then the node1 in Fig.8. is split with its prefix “ಕನ್ನಡ" (kannaDa) as a label of the root node (node1) and remaining length of the keys "ಬಾವುಟ" (baavuTa) as a subKey1 and “ಅಕ್ಷರ" (akShara) as subkey2 will become the labels of the children of the node1. Further details are of the V-KTPY PHT structure as shown in Fig. 8. and the insertion and searching algorithms are discussed in [5].

IV. EXPERIMENTAL SCENARIOS AND RESULTS

Our experiments are conducted with python implementation of V-KTPY encryption and VKTPY Family of Tries. The data set used for experiment are set of short political articles penned by noted journalist Shekhar Gupta for his ‘Prajavani’(A famous Kannada daily) column ’RaaShtrakaaraNa’[20]. These articles contains 6200 words after removal of stops words by our own tool [22]. The resulting text corpora can be found in [21].

Table-III. Details of the Data set

Data Set	*Total words in Doc w*	*Number of unique words u*	*Number of repeated words r*	*Number of stop words s*	*Total words4 w-s = u+r*
Prajavani articles	6200	2326	1465	2409	3791

Table III gives the details about the data set used. To create this data set the Kannada articles are downloaded from ‘Prajavani’ column and parsed in to tokens. Later they were encrypted by V-KTPY rule. These encrypted words are stored and searched in each of the family member of the V-KTPY Tries. The results, of several experiments with different V-KTPY Trie data structures are discussed under different sections.

A. Memory and Time taken by V-KTPY Trie and Kannada unicode Trie

The performance of the Simple Compressed V-KTPY Trie and V-KTPY 10Ary Trie are compared with simple Trie with Kannada unicode. The fig. 9. shows the memory usage of these Tries and it is clear from the figure that V-KTPY 10Ary out performs the rest two in saving the run time memory.

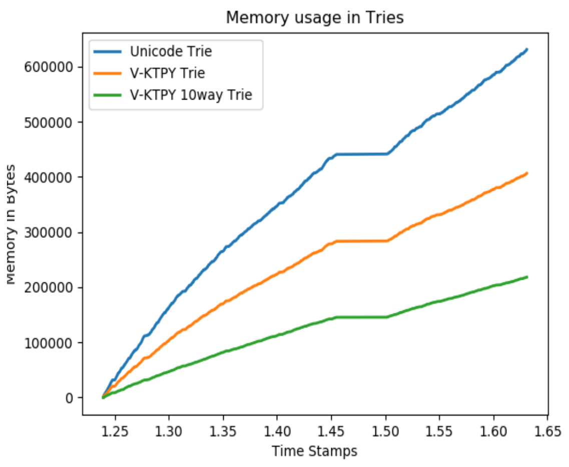

Fig. 9. Insertion: Memory vs Time taken by V-KTPY encoded and Kannada unicode Trie

Table-IV. Details of the Field links in Tries

Tries	*Simple unicode Trie*	*Proposed Simple compressed V-KTPY Trie*	*Proposed compressed variable length V-KTPY 10 Ary Trie*
Number of field links = (node + leaf)	11265	7255	3894

Table 4 gives the comparison of different field links (pointers) created with these three Tries.

This further proves that the number of fields links created by V-KTPY 10Ary Trie is very less compared to the other Tries ans thus it reduces the memory usage with good margin about 65%.

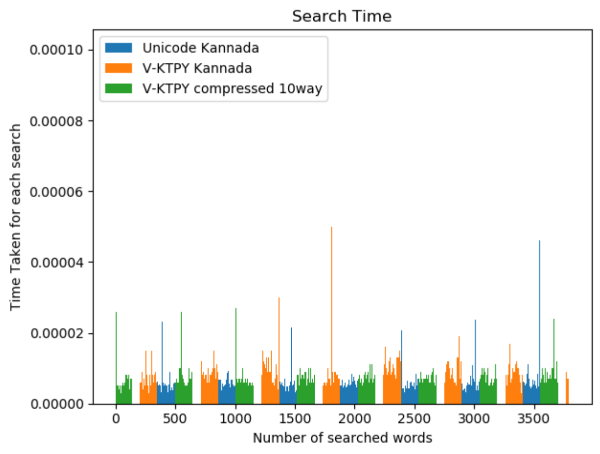

Figure 10: Search time taken by Unicode Kannada, V-KTPY encoded Kannada and V-KTPY encoded compressed 10 way Tries

Fig. 10 shows the comparison of average search time taken by these three Tries. This bar chart estimates the average time taken by these three Tries.

From this bar chart it is clear that the search time taken by these Tries are comparable and in an average the time complexity is same. This time trade off with the space is because it requires some time to construct conjunctive consonants from the search string.

A string with out any compound consonant takes less search time in V-KTPY 10Ary Trie compared to others two Tries.

B. Memory and time taken by Prefix hashed VKTPY PHT and fully compressed Kannada unicode **Trie**

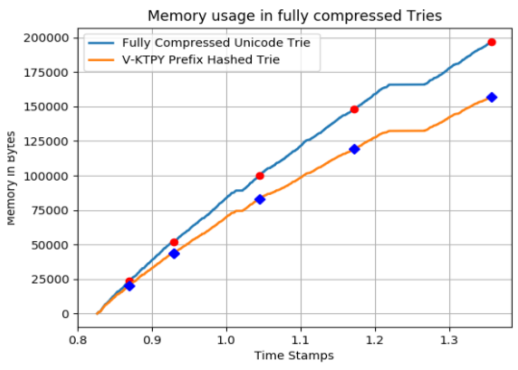

Fig. 11. **Insertion: Memory vs Time taken by VKTPY PHT and fully compressed Kannada unicode Trie** [5]

The VKTPY PHT is a fully compressed prefix hashed , sorted Trie. The performance of VKTPY PHT and a fully compressed Trie with unicode Kannada is plotted in Fig. 11. This figure shows VKTPY PHT gives a good compression and saves 20% of memory when compared with fully compressed unicode Kannada Trie. The details of the data set are discussed in [5].

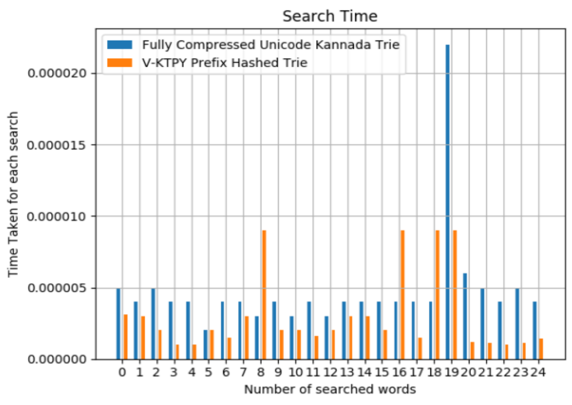

Fig. 12. Search time taken by fully compressed Unicode Kannada Trie and VKTPY PHT [5]

The comparison of search time taken by VKTPY PHT and fully compressed unicode Kannada Trie is plotted in Fig. 12. And on an average the time complexity of both tries looks the same.

V. CONCLUSIONS

The experimental results show that our proposed family of V-KTPY Tries are space efficient with almost same time complexity and suitable for indexing huge text corpora.

Our V-KTPY Trie gives 35% space efficiency, V-KTPY 10Ary Trie gives 65% space efficiency over simple unicode Trie. And the V-KTPY Prefix Hashed Trie gives 20% space efficiency when compared to fully compressed unicode Trie. The goal of this research is to use these results in Indian language modeling and the document retrieval tools. We would like to use V-KTPY numerical representation in Computational phonetic models, feature hashing and V-KTPY prefix hashed Tries in feature selection and extraction which would lead to efficient Kannada document representation. Further we would like to examine how V-KTPY codes work with LSTM a deep learning algorithm. This would be our future work with bigger text corpora.

REFERENCES

1. https://en.wikipedia.org/wiki/Brahmic_scripts
2. https://scriptsource.org/scr/Deva
3. https://en.wikipedia.org/wiki/Katapayadi_system
4. Subhash Kak, 2000, ”Indian binary numbers and the Katapayadi notation”, Annals of the Bhandarkar Oriental Research Institute, vol.81, 2000, pp.269-272
5. Yashaswini Hegde , Padma S. K, 2019, V-KTPY Prefix Hashed Trie for Indian Languages: A Case Study with Kannada Text Retrieval, INTERNATIONAL JOURNAL OF ENGINEERING RESEARCH & TECHNOLOGY (IJERT) Volume 08, Issue 06 (June 2019),
6. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.18.9659&rep=rep1&type=pdf
7. T.R.N. Rao and Subhash kak, August 8, 2016, ”Computation in Ancient India”, Paperback – August 8, 2016, 2017
8. Fredkin, E. 1960, ”Trie memory”, Communications of the ACM 3(9), 490–499.
9. Knuth, D. E., 1998, The Art of Computer Programming: Sorting and Searching, Vol. 3, second edn, Addison-Wesley.
10. Flajolet, P. & Puech, C. 1986, ”Partial match retrieval of multimedia
11. data”, Jour. of the ACM 33(2), 371–407.
12. Aoe et al. 1992 , ”An Efficient Implementation of Trie Structures”, SOFIWARE—PRACTICE AND EXPERIENCE,VOL.22(9), 695–721 (SEPTEMBER 1992)
13. Jon L. Bentley , Robert Sedgewick , 1997, ”Fast Fast algorithms for sorting and searching strings”, Proceeding SODA’97 Proceedings of the eighth annual ACM-SIAM symposiu on Discrete algorithms , Pages 360-369, Society for Industrial and Applied Mathematics Philadelphia, PA, USA ©1997
14. Bell, T. C., Cleary, J. G. & Witten, I. H. 1990, ”Text Compression”, Prentice-Hall. ISBN:0-13-911991-4
15. Robert Sedgewick, 1998 ,” Algorithms in C++, Parts 1–4: Fundamentals, Data Structure, Sorting, Searching”, Third Edition, Addison Wesley 1998, chapter 15.
16. M. Al-Suwaiyel and Ellis Horowitz, 1984, ”Algorithms for Trie Compaction”, ACM Trans. Database Syst., v.9, pp.243-263
17. Comer, D. 1979, ”Heuristics for trie index minimization”, ACM trans. on Database Systems 4(3), 383–395.
18. Acharya, A., Zhu, H. & Shen, K. 1999, ”Adaptive algorithms for cache-efficient trie search”,in ’Proc ALENEX Workshop on Algorithm Engineering and Experiments’, Springer-Verlag, pp. 296–311.
19. Heinz, S., Zobel, J. & Williams, H. E., 2002,”Burst tries: A fast, efficient data structure for string keys”, ACM trans. On Information Systems 20(2), 192–223.
20. Sumant Kulakari, Srinath Srinivasa ,”TrieIR: Indexing and Retrieval Engine for Kannada Unicode Text”,Digital Libraries: Social Media and Community Networks: 15th International Conference on Asia-Pacific Digital Libraries, ICADL 2013, Bangalore, India, December 9-11, 2013. Proceedings (pp.21-24)
21. http://www.prajavani.net/news/category/22885.html
22. https://drive.google.com/drive/folders/10MFrrFZLNhr7F0Ge9XOLgWeTgIxL2IyX
23. Y. Hegde, S. Kadambe and P. Naduthota, "Suffix stripping algorithm for Kannada information retrieval," 2013 International Conference on Advances in Computing, Communications and Informatics (ICACCI), Mysore, 2013, pp. 527-533.doi: 10.1109/ICACCI.2013.6637227

AUTHORS PROFILE

Yashaswini Hegde is a graduate in Electronics and Communications Engineering from UBDT College of Engg , Davangere. For 10 years she worked in various software companies and involved in several projects such as Telemetry Telecommand Data Base, Star tracking system (for ISRO) and E-commerce project based on Design Patterns in Sankhya Systems.

She also worked in the area of mobile applications such as Push-to-Talk and other client side applications for Sonim Technologies, Kodiak networks and Quallphone Inc. on different mobile OS. She completed her masters in Computer Science in 2009 from Mysuru university. She worked as Assistant professor in NIEIT Mysuru. She is pursuing her PhD under the guidance of Dr.S.K. Padma at VTU, Belagavi . She has published several research papers related to the development of Machine Learning techniques for Kannada Language.

Dr. S.K Padma is working as a Professor , Department of Information Science & Engineering , SJCE College Mysuru. She is a graduate in Electronics and Communication Engg. from SJCE Mysuru in 1984. She holds the master degree in Computer Engineering from SJCE Mysuru in 1992. She is a PhD degree holder for her works in the area of 'Computer and Information Sciences'. She has published many research papers and articles and a life member of Indian Society for Technical Education (ISTE) New Delhi:MISTE and Indian Institute of Engineers, India, MIE,M-124282-7. She also has membership of university and Institution authorities likes BOS and BOE. She has chaired many technical sessions of international conferences.