Knowledge Based Information Retrieval for Syntactic analysis of Kannada Script

Keshava Prasanna[1] Dr Ramakhanth Kumar P[2] Thungamani.M[3]ShravaniKrishna Rau[4]
[1,3]ResearchAssistant, Tumkur University, Tumkur
[2]Professor and HOD, R.V. College of Engineering, Bangalore
[4]Student, R.V. College of Engineering, Bangalore

Abstract
The output of even the most effectively designed OCR (Optical Character Recognition) module is not 100% accurate and hence errors occur in the identification of letters in turn leading to erroneous words. This motivates the use of spell checkers for syntactic analysis of the words which are output by the OCR and the need to verify the grammatical correctness of the sentences formed using the optional words. The use of spell checkers can be used to eliminate typographic errors and spell checkers form the heart of modern day Natural language Processing. An input word is taken from the user and it is searched for in a static data dictionary. The data dictionary is implemented using ternary search (TST) tree as the primary data structure.

Keywords
syntactical analyzer, ternary search tree, Levenstein distance, OCR, word recognition.

I. INTRODUCTION

The typical destination for documents is no longer assumed to be a hard copy. Increasingly, an electronic version is required for storage of a document. In order to achieve this, there arises a need for Optical Character Recognition (OCR) which can recognize the handwritten document and store it in an electronic form. However the output of the OCR module is not completely accurate.

There will be some erroneous data which can be corrected using word recognition. The word recognition module consists of a standard dictionary of a given script. Appropriate data structures are used to store the standard set of words in the dictionary. The erroneous words are searched one by one in the dictionary; if there is an inexact match the closest matches to that word are shown. This technique helps in increasing the efficiency with which the words are recognized in the OCR module. Word recognition can also be used to make the editors more intelligent by including features like auto completion and spell checking.

1.1 Methodology for word recognition

There are various techniques which are used in order to perform word recognition and spellchecking. Some of the techniques are as listed below

- Insertion
- Deletion
- Substitution
- Transposition

Insertion
Insertion is the technique wherein missing letter(s) are inserted into the input word at appropriate locations in such a way that the word matches a word in the dictionary.

Deletion
Deletion is the technique wherein letter(s) in the input word at appropriate locations is/are deleted in such a way that the word matches a word in the dictionary.

Substitution
Substitution is a technique wherein letter(s) in the input word is/are substituted with some other letter(s) in order to get a word that matches a word in the dictionary.

Transposition
As the name suggests transposition is the technique wherein the position of letter(s) in the input word is/are changed in such a way that the changed word conforms to a word in the provided dictionary.

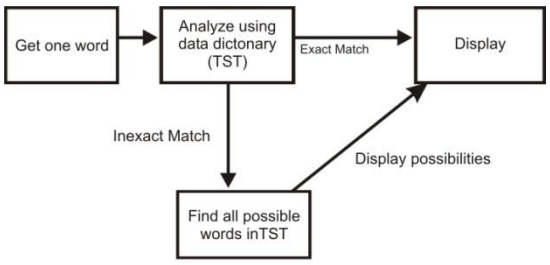

Figure 1.1 syntactical analyzer

II. WORD RECOGNITION

An input word is taken from the user interface module and it is searched for in a static data dictionary. The data dictionary is implemented using ternary tree as the primary data structure.

Manuscript received July 5, 2011
Manuscript revised July 20, 2011

2.1 The Kannada script

The Kannada alphabet is classified into two main categories [4]: vowels and consonants. There are 16 vowels and 35 consonants. Words in Kannada are composed of aksharas which are analogous to characters in an English word. While vowels and consonants are aksharas, the vast majority of aksharas are composed of combinations of these in a manner similar to most other Indian scripts.

An akshara can be one of the following,

- A stand alone vowel or a consonant
- A consonant modified by one or more consonants and a vowel.

2.2 Ternary search trees:

- Ternary search tree [3] is a data structure where in each node can have a maximum of 3 sons. it is similar to binary search tree except for the fact that if the letter being searched is equal to the letter in the current node then the search proceeds along the middle son
- Ternary trees provides a very space efficient solution but the time efficiency if *O(log m + n)* where m is the number of strings in the dictionary.
- The syntactical analyzer traverses the tree to find if the input is an exact match. If it does not find an exact match then it searches for closest possible words and provide all the possible alternatives in the order of non-increasing probabilities thus providing for spell checking.

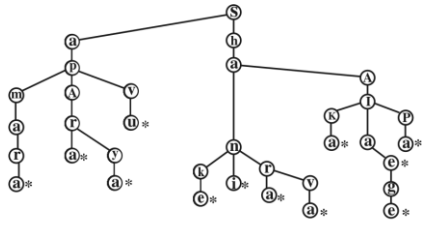

Figure 2.1: Example of a ternary tree

Example : ternary tree construction for words sara(ಸರ), , saradi(ಸರದಿ), saraLa(ಸರಳ), savi(ಸವಿ), samaya(ಸಮಯ), shani(ಶನಿ), shake(ಶಕೆ), shara(ಶರ), shava(ಶವ), shale (ಶಾಲೆ), shAke(ಶಾಕೆ), shAlege (ಶಾಲೆಗೆ)

2.3 Word Correction Strategies

- We use isolated word correction techniques i.e.unlike n-grams it is not context based
- We use the Levenshtein edit distance technique to define the distance between the given input word and the words which are displayed as options in order of non-increasing probabilities.

Levenshtein distance

- This is defined as the number of edit operations (insertion, deletion, substitution, transposition) required to convert from one string to the other
- For the case of checking OCR outputs the transposition technique is ignored since it is due to typographic error
- We display options which are at a Levenshtein edit distance of 2 or lesser and which have the same prefix as the input word.

Examples

* adhiKa (ಅಧಿಕ) is at an edit distance of one from adhika since the K is substituted with k.

* adhiKa (ಅಧಿಕ) is at an edit distance of 2 from both adhiKari (ಅಧಿಕಾರಿ) and adh Kara (ಅಧಿಕಾರ) since "ri" (i) and "ra" (ರ) 2 letters each have been added to the given word.

* So we give the option 'adhika' (ಅಧಿಕ) first followed by the other two options.

III. ALGORITHM DESCRIPTION

This part of the document is intended to provide an in depth working of the algorithms used in order to perform the required tasks. It provides an understanding of the data structures implemented.

3.1 Word processing

This is that module which does the initial word processing and error checking for words miss-spelt and providing the appropriate suggestions in order to correct the errors.

Data Structures Used

The main data structure used in this module is the ternary search tree which is used in order to implement the dictionary. The TST (Ternary Search Tree) is implemented as a linked list of nodes where in each node has the following structure:

Node This is the structure of each node of the ternary search tree. Each node has five fields.

Type: char Info	Type: Node * 1son	Type: Node * rson	Type: Node* mson	Type: bool IsCompleteString

Figure 3.1: Structure of node

This data structure is a linked list which is used as a part of the MList data structure. It is used in order to implement one of the inexact matching techniques namely space match. It has two fields: the index field and the pointer to the next node.

Type : int index	Type: Listnode* Nxt

Figure 3.2 : Structure of listnode

MList
This data structure is used in order to implement space match. It has only two fields the head and the tail field which are both of type Listnode pointer. It is used to keep track of the index in the input string where a valid word ends.

Type : Listnode* Head	Type : Listnode* tail

Figure 3.3: Structure of Mlist

TSTree
This is the primary data structure used. It is used in order to implement the dictionary. It has only one field the root.

Type: Node *Root

Figure 3.4: Structure of TST tree

Insertion into the Data Structure
The ternary search tree is implemented as a static data structure. All the nodes are inserted initially during the execution phase. The tree is created using the strings provided in the dictionary. The strings are sorted in an order in order to maximize efficiency (both space and time). The sorted string leads to the creation of the tree in an almost balanced fashion as a result of which the height of the tree is minimized thereby leading to minimal worst case search time. The average search time in the case of TST is ***O*** *(log (m) + n)* where m is the number of strings in the dictionary and n is the length of the pattern.

Algorithms Used
ReplaceMatch(in, lmatch, i, len)
This function replaces the characters in the input string in order to find a match in the dictionary. The inputs to this function are, input word(in), the node at which the match with the string is broken(lmatch), the number of characters matched exactly(i) and the length of the input string(len). The output is an array of possible strings

```
{
if( ( lmatch≠NULL) do
{
j←i;
strcpy(temp,in+i);
temp [0]←lmatch->info;
if(ExactMatch(temp,lmatch,i) do)
{
char temp1[30];
strcpy(temp1,i+1);
strcat(temp1,temp);
//push temp1 onto the array of output strings
array(outp,o++) ←temp1;
}
//recursively check the tree
replacematch(in,lmatch→rson,j,len);
replacematch(in,lmatch→lson,j,len);
}
}  //end of algorithm
```

This is the function that is used in order to replace a mismatching character at a time in order to find a corresponding match in the dictionary. The function works recursively in order to find a match in the dictionary. Taking the following example the replacement match strategy can be explained

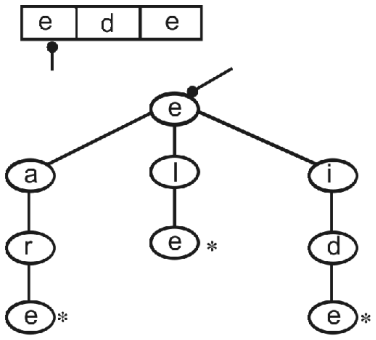

Figure 3.5a: Example TST for Replace match

As can be seen in the example above, the word "ede" is not a part of the dictionary thereby causing the exact match process to fail. Thus an attempt is made to find a approximate match to the word in the dictionary. The strategy used is replacement match (Substitution). The match is broken off at the character "d" of the word "ede". Thus a temporary array is initialized to "le" and this string is searched for in the dictionary "e" downwards. This happens as shown below:

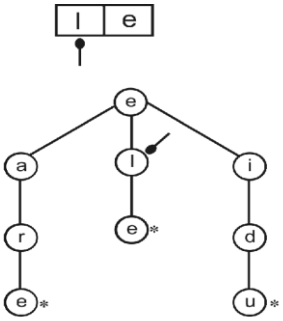

Figure 3.5b: Example TST for Replace match

As can be seen from the figure, a temporary array is initialized to "le" and it is searched in the tree "e" downwardIf an exact match occurs now, the two parts of the string are concatenated (namely "e" and "le") and added to the output. The replacement match then recursively continues along the left and right sons (if

present) in order to find matches. In this case the second node along the middle thread does not have any left and right sons so it stops here.

InsertMatch(in, lmatch, i, cnt, words)

This function inserts characters in the input string in order to find a match in the dictionary. The **input**s to this function are, input word (in), the node at which the match with the string is broken(lmatch), the number of characters matched exactly(i) and the Levinshtein distance(cnt).The output is an array of possible strings(words)

```
{
If ((Imatch≠NULL)) do
{
j←i, k←0;
array(temp,0)←lmatch→info;
array(temp,1)←0;
strcpy(temp+1,in+i);
strcpy(temp1,in);
array(temp1, j)←0;
if(cnt≠0)do
{
res ←Match(temp,cnt-1,tempoutp,lmatch);
//res indicates if the remaining string has been
//exactlymatched. 0 indicates exact match
if (res=0)dowords.concat(temp1,tempoutp);
//those with a distance of 1
else do words.concat (temp1,tempoutp,1);
//those with a distance of 2
}
else do
//edit distance of 2(both in the first part)
if(ExactMatch(temp,lmatch,i)) do
{
strcat(temp1,temp);
words.insert(temp1,1);
}
//recursively check the
treeinsertmatch(in,lmatch→rson,j,cnt,wordinsertmatch(i
n,lmatch→lson,j,cnt,words);
}
//end of algorithm
}
```

Now considering **insertion match**, the following example can be used in order to explain it:

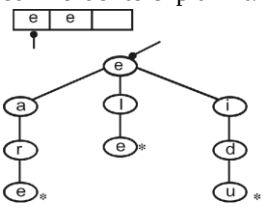

Figure 3.6a: Example TST for Insert Match

Now in this example, the match is broken off at the second "e". At this point a temporary array is initialized to "le" and the search for "le" continues "e" downward. Note that replacement match would initialize a temporary array with "l" (replacing the second "e") and this would fail.

The insertion match thread would then resemble the figure below

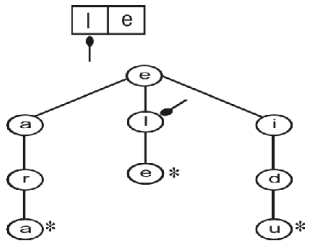

Figure 3.6b: Example TST for Insert Match

After completing the match for "le", the two parts ("e" and "le") are concatenated and then added to the output. The insertion match then recursively continues along the left and right sons (if present) in order to find matches. In this case the second node along the middle thread does not have any left and right sons so it stops here.

DeleteMatch(in, lmatch, i, cnt, words)

This function this deletes characters in the input string in order to find a match in the dictionary. The input to this function are, input word(in), the node at which the match with the string is //broken(lmatch), the number of characters matched exactly(i) and the Levinshtein distance(cnt).The output is an array of possible strings(words)

```
{
if( lmatch≠NULL)do
{
j←i;
strcpy(temp,in+i+1);
strcpy(temp1,in);
array(temp1, j)←0;
if(cnt≠0)do
{ res←Match(temp,0,tempoutp,lmatch);
//res indicates if the remaining string has been

//exactly matched. 0 indicates exact match
if(res≠0) words.concat(temp1,tempoutp);
//those with a distance of 1
else do
words.concat(temp1,tempoutp,1)
//those with a distance of 2
}
else if(Exact Match(temp,lmatch,i)) do
//those with a distance of 2, both in the first //part
{
strcat(temp1,temp);
```

```
words.insert(temp1,1);
}
}
}//end of algorithm}
```

Now considering **deletion match,** which can be explained with the following example?

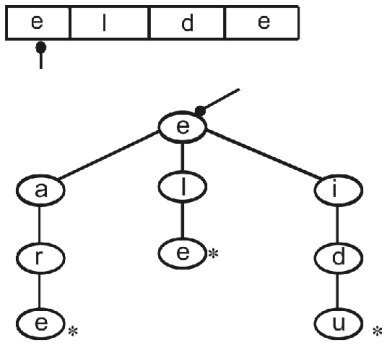

Figure 3.7a: Example TST for Deletion Match

As can be seen from the figure above the attempt to exactly match the input string "elde" with a string in the dictionary would fail. The two techniques described so far: Insertion matching and theSubstitution matching would fail. Delete match at tempts to find a match by deleting a character in the input string in order to find a match. A temporary array is initialized as shown below

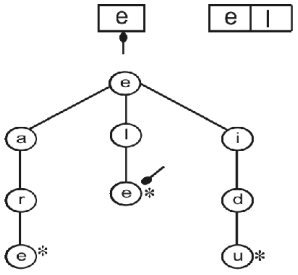

Figure 3.7b: Example TST for Deletion Match

The match initially breaks off at "d". This uses 2 temporary arrays: One with the prefix that has been matched exactly ("el" in our example) and another with the in format ion of the node where the match broke off ("e" in our example) as shown in the figure. Now when a match is found the two arrays are concatenated and the string is then added to the output. The deletion match then recursively continues along the left and right sons (if present) in order to find matches. In this case the second node along the middle thread does not have any left and right sons so it stops here.

spacematch(in, cnt, Nodes, words)

This function inserts space character in the input string in order to find a match in the dictionary.
The **input**s to this function are, input word(in), nodes is the linked list containing index positions,

where in each index represents the end of a valid string and the Levinshtein distance(cnt).

The output is an array of possible strings (words)

```
{
Iter←Nodes.head;
while(iter≠NULL ) do
{
i←0;
strcpy(temp,in+iter→index);
strcpy(temp1,in);
array (temp1, iter→index) ←' ';
array(temp1, iter→index+1)←0;
if(cnt=0) do
{
if(Exact Match(temp,root,i))do{
strcat(temp1,temp);
words.insert(temp1,1);
}
}
else do
{ res←Match(temp,0,tempoutp,root);
if(res≠0) do words.concat(temp1,tempoutp);
//edit distance of 1
else words.concat(temp1,tempoutp,1);
//edit distance of 2
}
iter←iter→nxt;//advance along the linked list
}
//end of algorithm
```

Transposition match (in, lmatch, i, cnt, words)
This function swaps two characters in the input string in order to find a match in the dictionary.

The inputs to this function are, input word(in), the node at which the match with the string is broken(lmatch), the number of characters matched exactly(i) and the Levinshtein distance(cnt).The output is an array of possible strings(words)

```
{
k←-1;
if( lmatch&& i<strlen(in)-1 )do
{
j ←i;
```

```
strcpy(temp,in+i);
  temp[0] ^=temp[1]^=temp[0]^=temp[1];
  strcpy(temp1,in);
  temp1[j] ←0;
  if(cnt) do
  {
  isExact←Match(temp,cnt-1,tempoutp,lmatch);
  //isExact indicates if the remaining string has been
  //exactlymatched. 0 indicates exact match
  if(isExact=1) do
  {
  wordsconcat(temp1,tempoutp,2);

  //those with edit distance 2
  }

  else if (isExact =0) do

            {
  words.concat(temp1,tempoutp,1);
  //those with edit distance 1
  }
  }
  else if(ExactMatch(temp,lmatch,i))do
  {
  strcat(temp1,temp);
  words.insert(temp1,1);
  k ←1;
  }
  }//end of algorithm
```

Now considering **transposition match** which can be explained with the following example:

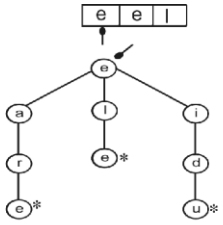

Figure 3.8a: Example TST for Transposition Match

Now in this example, the match is broken off at the second "e". At this point a temporary array is Initialized to "le" and the search for "le" continues "e" downward. Note that replacement match would initialize a temporary array with "l" (replacing the second "e") and this would fail .The transposition match thread would then resemble the figure below

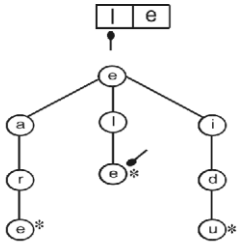

Figure 3.8b: Example TST for Transposition Match

IV. EXPERIMENTALRESULTS

Input : aKula (ಅಕುಲ)
Expected Output : aKila (ಅಖಿಲ)
Observed Output : aKila (ಅಖಿಲ)
Remarks : The word aKula (ಅಕುಲ) is not found in the dictionary, the match ends at aK, after that the letter u is replaced by all the options, in this case it is letter i , leading to aKila (ಅಖಿಲ)

Input : horadu (ಹೊರದು)
Expected output : ಹೊರತು
ಹೊರಸು
ಹೊರಳು

Observed output : (ಹೊರತು)
(ಹೊರಸು)
(ಹೊರದು)
(ಹೊರಳು)

Remarks : The word horadu is not found in the dictionary the match ends at hora. After that the replacement strategy tries to replace 'd' with the available options which in this case are 's','t','D' and 'L' leading to the corresponding options horatu (ಹೊರತು), horasu (ಹೊರಸು), horaDu (ಹೊರದು) and horaLu (ಹೊರಳು).

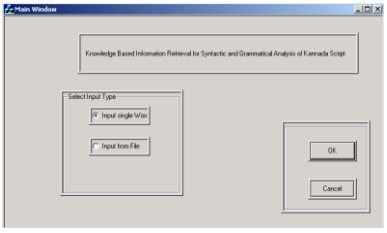

Figure 4.1 Main Window

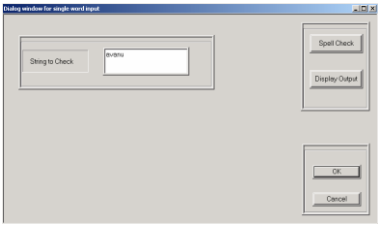

Figure 4.2 Window for single

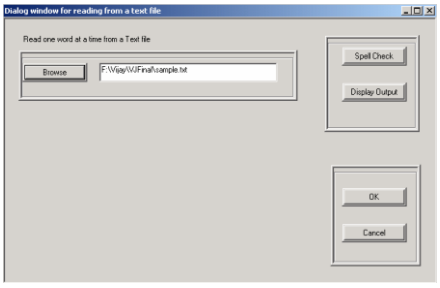

Figure 4.3 window for reading from a text file

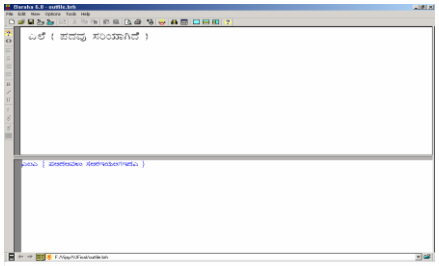

Figure 4.4 Output in Baraha for Exact Match

CONCLUSION

In the published form each TST node requires three pointers, a split character and the associated value. It has been [2] shown that the cost of the algorithm is proportional to *lgN*byte comparisons where *N* is the number of keys in the dictionary. In practice a TST seems to perform better than this would suggest the cached memory hierarchy of modern computers and the skew in the distribution tree branches give enhanced performance. For a moderate size dictionary [3], up to 50000 keys, the performance of a TST is excellent and almost independent of the number of keys. However, for very large dictionaries or where main memory performance approaches the cache speed or where the dictionary is only infrequently referenced from a larger application then the *lgN*performance becomes apparent. The performance of a TST, as with simple binary trees, can be degraded by the same degenerate case of inserting keys in order, instead of the benefits of a *lgN*search at each trie branch it can degenerate to an *N*=2 search, where *N* in this case has a maximum of the alphabet cardinality. For the performance comparisons, tree balancing was added to the TST insert function.

REFERENCES

[1] Bentley, J. and Sedgewick, R. 1997. Fast algorithms for sorting and searching strings.In Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (1997), SIAM Press (1997).

[2] Clement, J., Flajolet, P., and Vallee, B. 1998. The analysis of hybrid trie structures. In Proceedings of Western Joint Computer Conference, Volume Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (1998), pp.531–539. SIAM Press.

[3] Bagwell, Phill "Fast and Space efficient trie searches",2003

[4] T V Ashwin and P S Sastry, "A font and size-independent OCR system for printed Kannada documents using support vector machines", Sadhana Vol. 27, Part 1, February 2002, pp. 35–58. © Printed in India

[5] Beaza Yates, Ricardo, Gonzalo Navarro "Fast approximate string matching in dictionary", 2004

[6] Dan Gusfield, "Algorithms on Strings, Trees, and Sequences", First South Asia Edition 2005.

KeshavaPrasanna received B.E from Bangalore University and M.Tech in Information and Technology in the year 2005.He has experience of around 13 years in academics. Currently pursuing Ph.D. and working as Research Assistant in Tumkur University, Tumkur. Life membership in Indian Society for Technical Education (ISTE).

Dr.Ramakanth Kumar P completed his Ph.D. from Mangalore University in the area of Pattern Recognition. He has experience of around 16 years in Academics and Industry. His areas of interest are Image Processing, Pattern Recognition and Natural Language Processing. He has to his credits 03 National Journals, 15 International Journals, 20 Conferences. He is a member of the Computer Society of India (CSI) and a life member of Indian Society for Technical Education (ISTE). He has completed number of research and consultancy projects for DRDO.

Thungamani.M received B.E from Visvesvaraya Technological University and M.Tech in Computer Science and Engineering in the year 2007.She has experience of around 08 years in academics. Currently pursuing Ph.D. and working as Research Assistant in Tumkur University, Tumkur. Life membership in Indian Society for Technical Education (MISTE) The Institution of Electronics and Telecommunication Engineers (IETE).

Shravani Krishna Rau pursuing B.E degree in Computer Science & Engineering from Visvesvaraya Technological University. Student ofR.V.College of Engineering, Bangalore.