CHAPTER V
DICTIONARY DESIGN AND SEARCHING

Introduction

This chapter will discuss a novel and effective method of designing and searching of MMD (Multilingual Manipuri Dictionary). The main objective of this chapter is to study how to design model of user-friendly interface that exploits multilingual information to its full potential using Meitei Mayek as target language and Hindi as well as English as pivot language. Being a Multilingual dictionary, MMD has fewer limitations and the description of language specifications of the headword in each entry of dictionary is simple and simultaneously more comprehensive. Considering the needs of the translator, a language researcher or and everyday user, this chapter will make effort to present the best design, best algorithm for dictionary searching and searching methods in MMD. All tools and data in MMD will be made freely and publicly available after launching the site.

5.1 System Design

In Machine Translation (MT) system, Dictionary is the most important since it decides the purchasing power of the translation system. It should contain all the information necessary for the computer to understand and use the natural language. There are different approaches possible for dictionary design. Also there may be different levels of dictionaries like stem/root word dictionary, grammatical information dictionaries etc. Dictionary can be incorporated into three types of knowledge, which are

a) Linguistic knowledge

b) Context knowledge and

c) Specialized knowledge(expertise).

Components of linguistic knowledge are

I. Knowledge about nouns representing concepts and verbs representing actions.

II. Morphological knowledge determining the formations of words.

III. Syntactical knowledge determining the function between grammatical elements.

IV. Semantic knowledge indicating the concerned relationship between words(noun and verbs).

Components of contextual knowledge:

I. Knowledge of linguistic context expressing the relationship between preceding and following sections of a discourse and a text.

II. Knowledge about over context determining the situations and scenes at a particular moment.

III. Knowledge about the psychological context determining the level of conciousness between a speaker and listener at the moment of speaking.

Components of specialized knowledge (expertise):

I. Concept between conceptual objects, hyper concept or hypo concepts.

II. Partial and full relationship between conceptual object.

III. Association relationships between conceptual object.

For different knowledge, different types of dictionaries are possible. Researches on the conceptual and expertise knowledge are new approaches and there is yet not any proven way to incorporate them into an actual system. Finding means to store all these different knowledge, to facilitate their retrieval, to use and integrate and to combine all the various processing methods in a total knowledge system in general and machine translation research and development in particular. The problem is to find out a data structure to store all possible words of the dictionaries with the minimum possible storage space and a fast and easy retrieval algorithm. A popular method to reduce the dictionary size is to divide the words into stems (root words) and ending (suffixes) or beginning (prefixes). Only the stem or root is stored since it is wasteful to include every inflected form of nouns and verbs.

Of course, one must be careful that the meaning of each word is properly determined by its stem and suffixes and/or prefixes; if not, the exceptions should be

entered into the dictionary so that they will be found before we attempt to look for a suffix and/or prefix.

However, all the words that are occurring in source language text cannot always be included in dictionaries. Basically there are two approaches, either to attempt some kinds of analysis and translation or to print out the unallocated source language form. In both the cases there is a further problem with the rest of the sentence, whether to attempt an incomplete translation or to give up and produce no translation. In experimental MT system, it is desirable to produce some kind of translation. Some of source language words may correspond to a number of different target language words. The translation may print out all the possibilities or it may attempt to select the one, which is most appropriate for the specific text in source language.

Homonyms are words, which have two or more distinct and unrelated meaning and features. For examples in English language, words like 'bank' has different meaning representing 'geological feature' as well as 'financial institute'. In Manipuri language, also words like ꯋꯥ(waa) has different meaning representing 'a bamboo plant' as well as 'speech spoken by some person'. Homophones are words, which sound the same but have different meaning. Homographs are words, which are spelled the same but different meaning such as 'tear (crying and ripping)' in English and 'ꯀꯥꯕꯥ (Kaabaa)' in Manipuri. In MT system the homophone problem is irrelevant since only written text is considered.

For practical purpose it is immaterial whether the Source Language word is a homograph or polyseme; the problem for MT is the same, the relevant meaning for the context is to be identified and the appropriate Target Language form must be selected, which is called as 'homograph resolution' in MT. Sometimes the Target Language vocabulary makes finer sense distinctions than the source language which are tackled by homograph resolution and may use contextual knowledge to make appropriate selection.

The figure given below shows how root words or headwords of different categories and prefix/suffixes can be combine together to form the maximum numbers of words for storage in MMD.

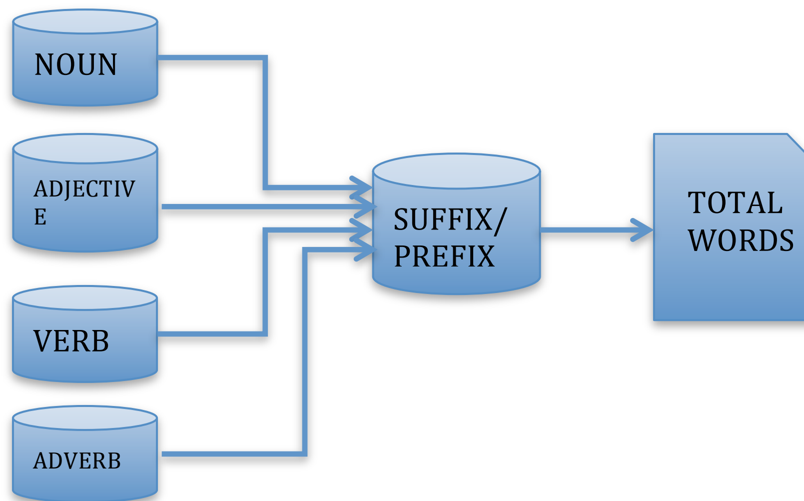


Figure 41: Words formation with root word and affixes

5.2 Sequential Structure

In this structure, the words in the dictionary are stored sequentially. Every word can have fixed length or may be followed by a separator marker. A pointer may be associated with each word pointing to separate area in the dictionary containing grammatical or other relevant information. Otherwise this information may be associated along with the words also. Searching time in the dictionary is higher when words are followed by a marker since only linear search techniques can be applied that can be approved by hashing methods and associated pointers as the words are sorted. Searching time is lesser when words are of fixed length since a modified binary search method can be applied. The storage requirements in the two cases are comparable as the fixed word length scheme has a number of unused positions but not explicit pointers while the variable word length scheme has no unused position but pointers are included.

In the sequential structure of dictionary instead of storing every inflected forms of noun, verb etc. only the base word is stored and prefixes or suffixes may be specified as associated information. This is the same as words appear in the commonly available dictionaries. Further, improvement is possible if a character, which appears at the same position of a number of consecutive words, then the character can be stored only once and appropriate points may be used to point to the various words that are possible. Extra space will be required for storing the additional pointers but number of

comparisons involved during the searching of a word is proportional to the actual word length. In the sequential structure with fixed word length the number of comparisons involved during searching of words is proportional to the product of the logarithm of the number of entries and word-length. This improved storage method requires a trie structure which store the entries.

5.3 Trie Structure

The trie of a dictionary can be defined as follows:

1. The root node is null.
2. The first letter of any word appears at the level below the root.
3. Each node on level '1' stores a character which is common to a set of stems that begin with a certain sequence of 1 character. The node specifies an M-way (M<=40) branch depending on the (i+1)th characters.
4. Besides the root nodes, there are three other types of nodes depending on the type of pointers associated with the node in addition to the character.
   a) Leaf node – contains a pointer to the grammatical information file.
   b) Internal node – contains a pointer to the text sibling as the same level.
   c) Both internal and leaf node – contains both the above information.

At any level if there is no branching from a node then the remaining letters of the stem are stored consecutively at the node followed by a pointer to the grammatical information file

Table 19: Table of trie for English word

1. a	5. accent
2. aback	6. acid
3. ability	7. add
4. above	8. addict
5. acacia	9. address

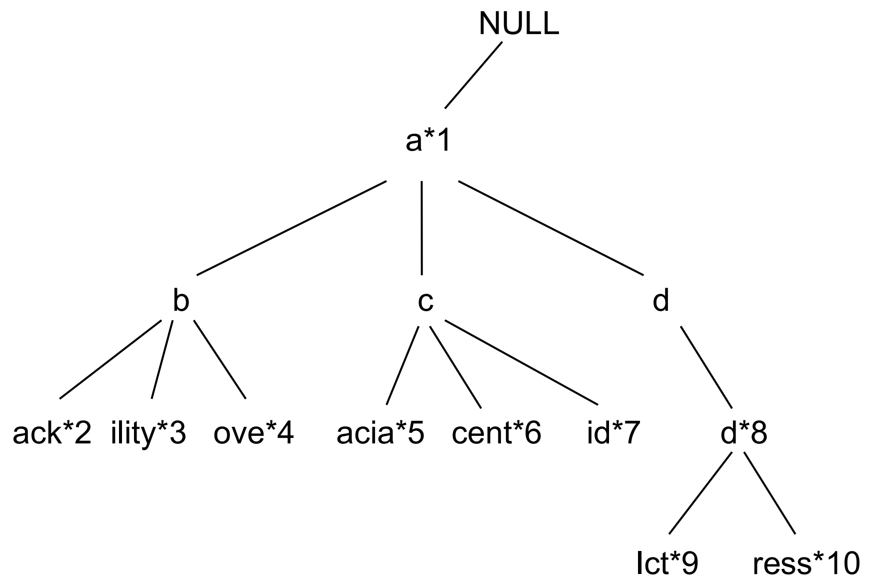


Figure 42: A trie structure of English words with reference to table no. 19

The pre-order traversal of this trie is stored in the trie dictionary. The dictionary for these 10 words would be as sown in the table number 20.

Table 20: Example trie in English

a---------*1 b------a5---ck*2i7---lity*3ove*4

c---a6---cia*5c6---ent*6 id *7

d----d----*8 i6---ct*9 ress*10

Trie structure of the of Manipuri words like Achaabaa, Achoubaa, Akaaba, Akhaabaa, Angaangbaa, Angoubaa, Apikpaa, Apubaa, Athaabaa and Athiibaa is given below in the figure no 45.

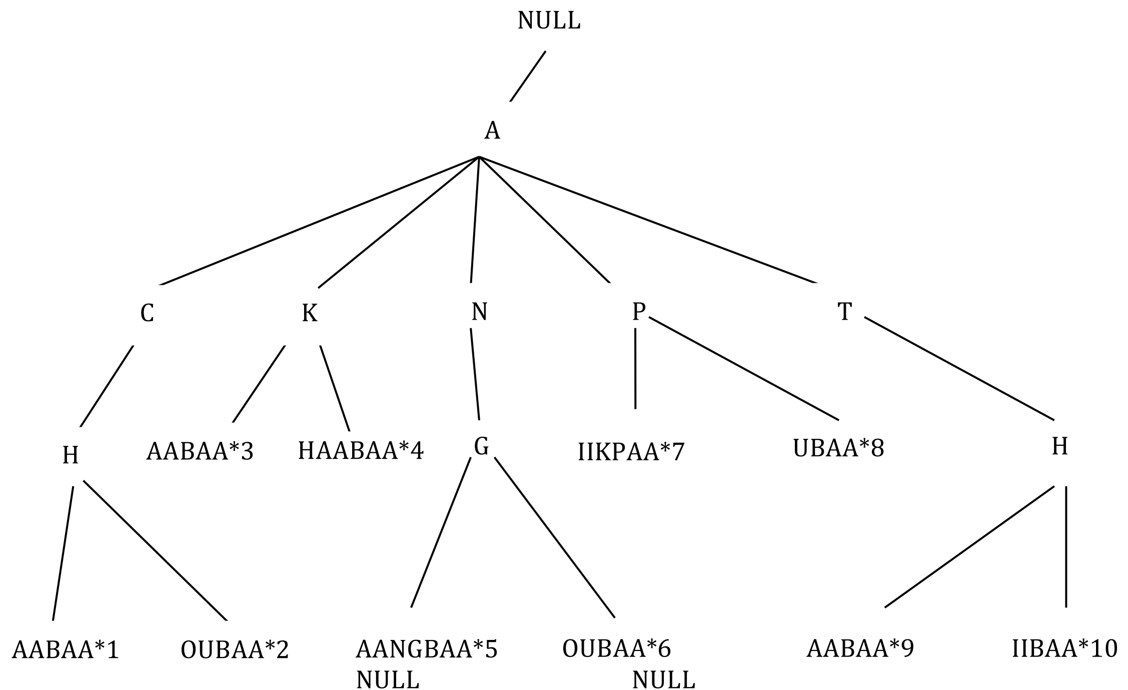


Figure 43: A trie structure of Manipuri

Table 21: Example trie in Manipuri

A------C—H---A7-ABAA*1 OUBAA*2

K—A7—ABAA*3 HAABAA*4

N—G---A9-ANGBAA*5 OUBAA*6

P—18—IKPAA*7 OUBAA*8

T-----H---A7-ABAA*9 IIBAA*10

Some pointer address spaces do not get filled up (denoted by '-') since there is no sibling character at the same level for those characters in the example trie.

5.3.1 Trie Search Algorithm

The search algorithm involves the following steps

1. For each character in the string, see if there is a child node with that character as the content.
2. If that character does not exist, return false
3. If that character exists, repeat step 1.

4. Do the above steps until the end of string is reached.

5. When end of string is reached and if the marker of the current Node is set to true, return true, else return false.

5.3.2 The Algorithm of The Trie Generation

1. Initialize the buffer and the toggle variables, different counters and related variables.
2. For each stem in the dictionary do repeat step 3 to 7.
3. Read each character of the stem word until the end of the word is encountered.
4. For the ith character do repeat steps 5 to 7.
5. (a) if the ith character is occurring for the first time i.e. not equal to the ith character of the previous stem scanned, the character is put in the trie. Increment appropriate siblings address counters.

   (b)if toggle[PREV]=FALSE, do nothing

Else check the position of this character with the mark position (a variable defined later in the algorithm). POP from the stack (containing starting point of reserved spaces, defined later) until the starting of the reserved space for the present i^{th} position is retrieved.

The value of the ith sibling address counter is written from the position popper from the stack. Initialize this and all higher indexed sibling address counter to ZERO. Set toggle [PREV]=FALSE.

(c) if the character of the next stem is same as the ith character for the current stem, reserve the required amount of space for the sibling pointer. The starting point of this reserved space is pushed into a stack and the sibling address counter for the position 'I' is set to ZERO. The size of the sibling pointer depends on the value of 'I' i.e. the character position. The smaller the value of 'I' the larger will be the size as there will be more entire in the trie between any two siblings at the ith level then at the $(i+1)^{th}$ level. The reserved spaces will be filled up when the sibling is obtained for the present character. Increase the value of 'i' and repeat steps 5(b) and 5(c) for the ith character.

ELSE if the ith character of the next word is different (i) mark toggle [CURR]=TRUE, (ii) the current value of ‘i’ is set as the mark position and (iii) the required amount of space is reserved in the trie, its starting position put in the stack and the ith counter is set to ZERO. Thereafter put the remaining characters one after another in the trie until the end of the stem is encountered. Increment appropriate siblings address pointers each time a character is entered. Go to step 7.

6. If the ith character is already entered into the trie (i.e. same as the ith character of the previous stem) do

   (a) If the ith character is same as the ith characters of the next stem, increment ‘i’ and go to step 4.

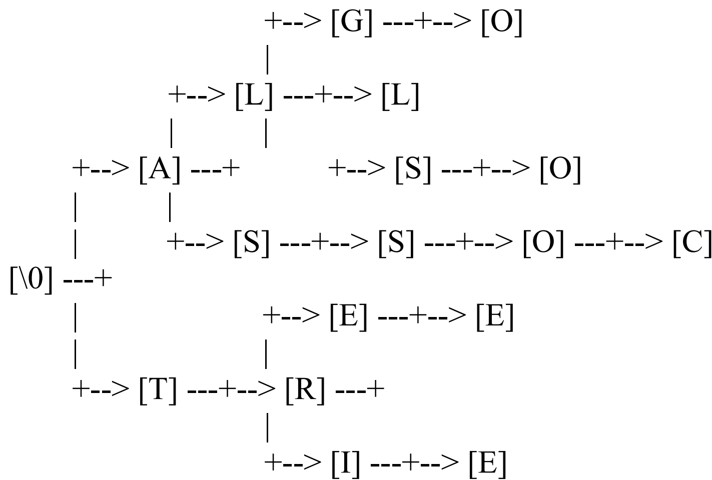
   (b) ELSE if the ith is different with the ith character of the next word, ‘i’ mark together [CURR]=TRUE and (ii) the current value of ‘i’ is set as the mark position. Thereafter check only with the previous stem till the end of the current stem is encountered. Once a difference occurs at a particular character position, put this character in the trie. Repeat step 5(b).Thereafter write the remaining character in the trie consecutively till the end of the current stem. Increment appropriate sibling address counters each time a character is enter in this trie.

7. Put a ‘*’ in the trie, followed by the stem pointer. Increment the current stem pointer value. Update the buffer if toggle [CURR]=TRUE set toggle[CURR]=FALSE and toggle[PREV]=TRUE.

8. Close all the files an exit.

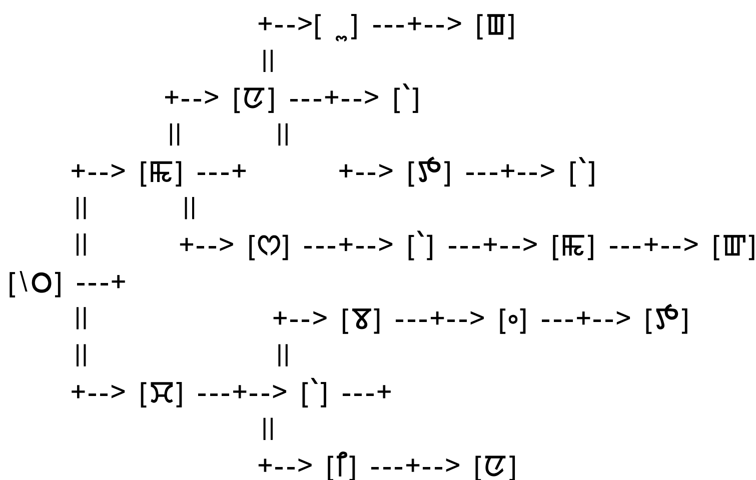
The trie file so created will have same unused blank spaces, as some of the sibling address positions do not get filled up.

Sample Data (English): - ALGO, ALL, ALSO, ASSOC, TREE, TRIE



*/

Sample Data (Manipuri) [illegible]



*/

5.3.3 Advantages of Tries over Binary Search Trees (Bsts)

1. Looking up keys is faster. Looking up a key of length *m* takes worst-case *O(m)* time. A BST performs *O (log(n))* comparisons of keys, where *n* is the number of elements in the tree, because lookups depend on the depth of the tree, which is logarithmic in the number of keys if the tree is balanced. Hence in the worst case, a BST takes *O* (*m* log *n*) time. Moreover, in the worst case log (*n*) will approach *m*. Also, the simple operations tries use during lookup, such as array indexing using a character, are fast on real machines.

2. Tries are more space-efficient when they contain a large number of short keys, since nodes are shared between keys with common initial subsequences.

3. Tries facilitating longest-prefix matching.

4. The number of internal nodes from root to leaf equals the length of the key. Balancing the tree is therefore of no concern.

5.3.4 Advantages of Tries over Hash Tables

1) Tries support ordered iteration, whereas iteration over a hash table will result in a pseudorandom order given by the hash function (and further affected by the order of hash collisions, which is determined by the implementation).
2) Tries facilitate longest-prefix matching, but hashing does not, as a consequence of the above. Performing such a "closest fit" find can, depending on implementation, be as quick as an exact find.
3) Tries tend to be faster on average at insertion than hash tables because hash tables must rebuild their index when it becomes full, a very expensive operation. Tries therefore have much better bounded worst-case time costs, which is important for latency-sensitive programs.

   Since no hash function is used, tries are generally faster than hash tables for small keys.

5.3.5 Complexity Analysis

Now that we've seen the basic operations on how to work with a TRIE, we shall now see the space and time complexities. To measure the complexity we considered two important operations INSERT and SEARCH. Lets us always take into account the worst-case timing first and later convince us of the practical timings. For every Node in the TRIE we had something called as Collection where the Collection can be either a Set or a List. If we choose Set, the order of whatever operation we perform over that will be in $O(1)$ time, whereas if we use a Linked List the number of comparisons at worst will be 40 (the number of alphabets). So for moving from one node to another, there will be at least 40 comparisons, which will be required at each step.

Having these in mind, for inserting a word of length 'k' we need ($k * 40$) comparisons. By Applying the Big *O* notation it becomes $O(k)$, which will be again O (1). Thus insert operations are performed in constant time irrespective of the length of

the input string (this might look like an understatement, but if we make the length of the input string a worst case maximum, this sentence holds good).

It holds good for the search operation as well. The search operation exactly performs the way the insert does and its order is $O(k*40) = O(1)$.

5.4 Suffix Tree

In computer science, a **suffix tree** (also called **PAT tree** or, in an earlier form, **position tree**) is a compressed trie containing all the suffixes of the given text as their keys and positions in the text as their values. Suffix trees allow particularly fast implementations of many important string operations.

The construction of such a tree for the string *S* takes time and space linear in the length of *S*. Once constructed, several operations can be performed quickly, for instance locating a substring in *S*, locating a substring if a certain number of mistakes are allowed, locating matches for a regular expression pattern etc. Suffix trees also provided one of the first linear-time solutions for the longest common substring problem. These speedups come at a cost: storing a string's suffix tree typically requires significantly more space than storing the string itself.

The string obtained by concatenating all the string-labels found on the path from the root to leaf i spells out suffix $S[i..n]$, for i from 1 to n.

Since such a tree does not exist for all strings, *S* is padded with a terminal symbol not seen in the string (usually denoted $). This ensures that no suffix is a prefix of another, and that there will be ***n*** leaf nodes, one for each of the n suffixes of *S*. Since all internal non-root nodes are branching, there can be at most $n - 1$ such nodes, and $n + (n - 1) + 1 = 2n$ nodes in total (n leaves, $n - 1$ internal non-root nodes, 1 root). An example of Manipuri using English text of suffix is given below as for Manipuri one has to classify the suffix in the API and root word with proper grammar:

Chatkhrabada (Have gone/ have left)

8: a
4: abada
6: ada
5: bada
7: da
2: hrabada
1: khrabada
3: rabada

```
 |(1:khrabada)|leaf
tree:|
 |(2:hrabada)|leaf
|
|(3:rabada)|leaf
|
|    |(5:bada)|leaf
|(4:a)|
|    |(7:da)|leaf
 |
 |(5:bada)|leaf
|
|(7:da)|leaf
2 branching nodes
```

5.4.1 Functionality

A suffix tree for a string *S* of length *n* can be built in *θ (n*) time, if the letters come from an alphabet of integers in a polynomial range (in particular, this is true for constant-sized alphabets)(Farach, Martin, 1997) For larger alphabets, the running time is dominated by first sorting the letters to bring them into a range of size *0(n log n);* in general, this takes *0(n log n)* time.

Assume that a suffix tree has been built for the string *S* of length *n*, or that a generalized suffix tree has been built for the set of strings *D={S1.S2….Sk}* of total length $n = |n_1| + |n_2| + \cdots + |n_K|$ we can:

5.4.2 Search for Strings

1. Check if a string *P* of length *m* is a substring in *0(m)* time(Gusfield, Dan, 1999).
2. Find the first occurrence of the patterns $P_1, ...P_q$ of total length *m* as substrings in *0(m)* time.
3. Find all z occurrences of the patterns $P_1, ...P_q$ of total length *m* as substrings in *0(m+z)* time(Gusfield, Dan, 1999).
4. Search for a regular expression *P* in time expected sublinear in ***n***.(Baeza-Yates, et al., 1996)

5. Find for each suffix of a pattern *P*, the length of the longest match between a prefix of *P[i…m]* and a substring in *D* in $\theta(m)$ time(Gusfield, Dan, 1999) This is termed the matching statistics for *P*.

5.4.3 Find Properties of the Strings

1. Find the longest common substrings of the string S_i and S_j in $\theta(n_i+n_j)$ time(Gusfield, Dan, 1999).
2. Find all maximal pairs, maximal repeats or supermaximal repeats in $\theta(n+z)$ time(Gusfield, Dan, 1999).
3. Find the Lempel–Ziv decomposition in $\theta(n)$ time(Gusfield, Dan, 1999).
4. Find the longest repeated substrings in $\theta(n)$ time.
5. Find the most frequently occurring substrings of a minimum length in $\theta(n)$ time.
6. Find the shortest strings from Σ that do not occur in *D*, in $0(n+z)$ time, if there are Z such strings.
7. Find the shortest substrings occurring only once in $\theta(n)$ time.
8. Find, for each ***i***, the shortest substrings of S_i not occurring elsewhere in *D* in $\theta(n)$ time.
9. The suffix tree can be prepared for constant time lowest common ancestor retrieval between nodes in $\theta(n)$ time(Gusfield, Dan, 1999).One can then also:
10. Find the longest common prefix between the suffixes $S_i[p..n_i]$ and $S_i[p..n_j]$ in $\boldsymbol{\theta(1)}$ (Gusfield, Dan, 1999).
11. Search for a pattern *P* of length *m* with at most *k* mismatches in $0(Kn+z)$ time, where *z* is the number of hits(Gusfield, Dan, 1999).
12. Find all ***Z*** maximal palindromes in $\theta(n)$, or $\theta(gn)$ time if gaps of length ***g*** are allowed, or $\boldsymbol{\theta(Kn)}$ if ***K*** mismatches are allowed(Gusfield, Dan, 1999).
13. Find all ***Z*** tandem repeats in $0(nlog\ n + Z)$, and *k*-mismatch tandem repeats in $0(Kn\ log(n/K)+ Z)$ (Gusfield, Dan, 1999).
14. Find the longest substrings common to at least *K* strings in *D* for $K=2,…,K$ in $\boldsymbol{\theta(n)}$ time(Gusfield, Dan, 1999).
15. Find the longest palindromic substring of a given string (using the suffix trees of both the string and its reverse) in linear time(Gusfield, Dan, 1999).

5.4.4 Applications

Suffix trees can be used to solve a large number of string problems that occur in text-editing, free-text search, computational biology and other application areas. (Zamir, Oren; Etzioni, Oren, 1998) Primary applications include:(L. Allison, 2014)

1. String search, in *O(m)* complexity, where *m* is the length of the sub-string (but with initial *O(n)* time required to build the suffix tree for the string)
2. Finding the longest repeated substring
3. Finding the longest common substring
4. Finding the longest palindrome in a string

A suffix tree is also used in suffix tree clustering; a data-clustering algorithm used in some search engines (Zamir, et al, 1998).

5.5 Implementation

If each node and edge can be represented in $\theta(1)$ space, the entire tree can be represented in $\theta(n)$ space. The total length of all the strings on all of the edges in the tree is $\theta(n^2)$, but each edge can be stored as the position and length of a substring of *S*, giving a total space usage of $\theta(n)$ computer words. The worst-case space usage of a suffix tree is seen with a Fibonacci word, giving the full 2n nodes.

An important choice when making a suffix tree implementation is the parent-child relationships between nodes. The most common is using linked lists called **sibling lists**. Each node has a pointer to its first child, and to the next node in the child list it is a part of. Other implementations with efficient running time properties use hash maps, sorted or unsorted arrays (with array doubling), or balanced search trees.

5.6 MMD Design

As the MMD is web enabled the context diagram of MMD is given below

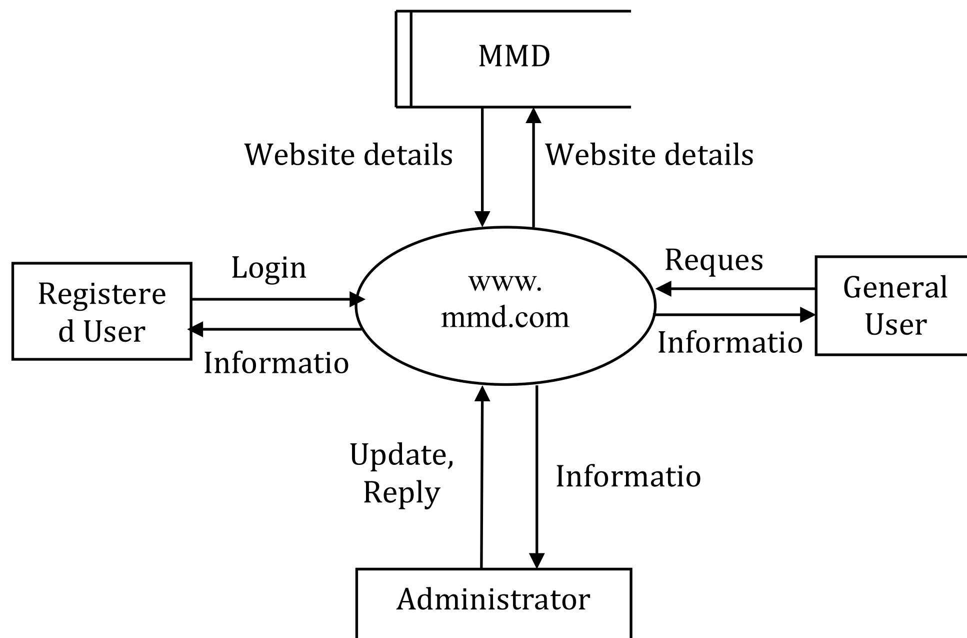


Figure 44: Context diagram for www.mmd.com

5.6.1 Page Hierarchy of MMD

The hierarachy form of the pages of MMD is organised in two modules one for Adminitrative side and one for user side. Administrator has right to edit and update the data and information to the site. And user can view or one can register for sending data and information to the Admin.

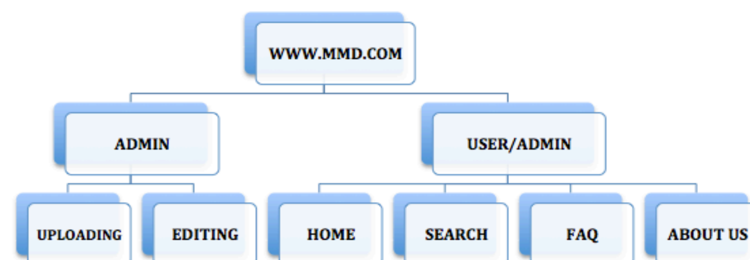


Figure 45: Hierarchical page chart of MMD

5.7 Process Diagram of MMD

The figure given below shows how to store data or lexicon in MMD database from different sources like books, Dictionary and corpus.

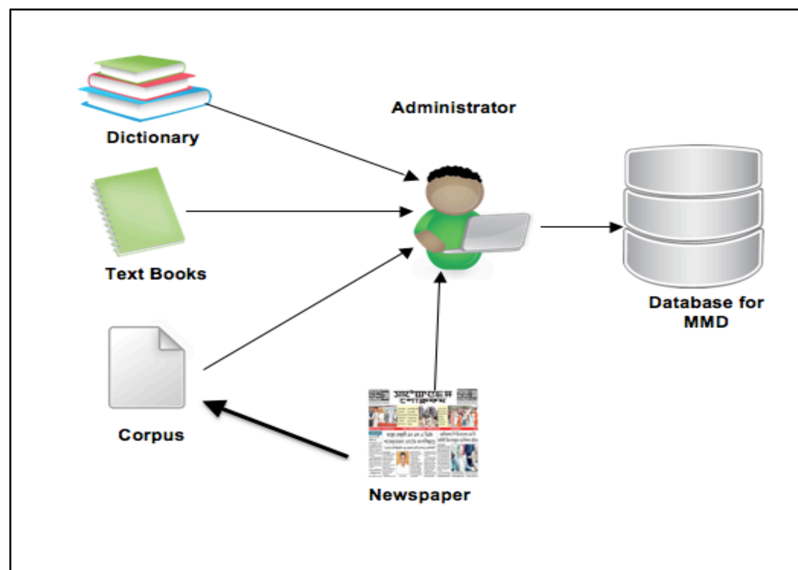


Figure 46: Process Diagram of MMD database

5.8 Process Diagram of SL And TR Language

The process of parallel working with English, Hindi language with Manipuri language is showing in the given below diagram. In this preocess whle translating from source language we did translation of words , transliteration and in last methods we coined the words in target language if did not exist in target language.

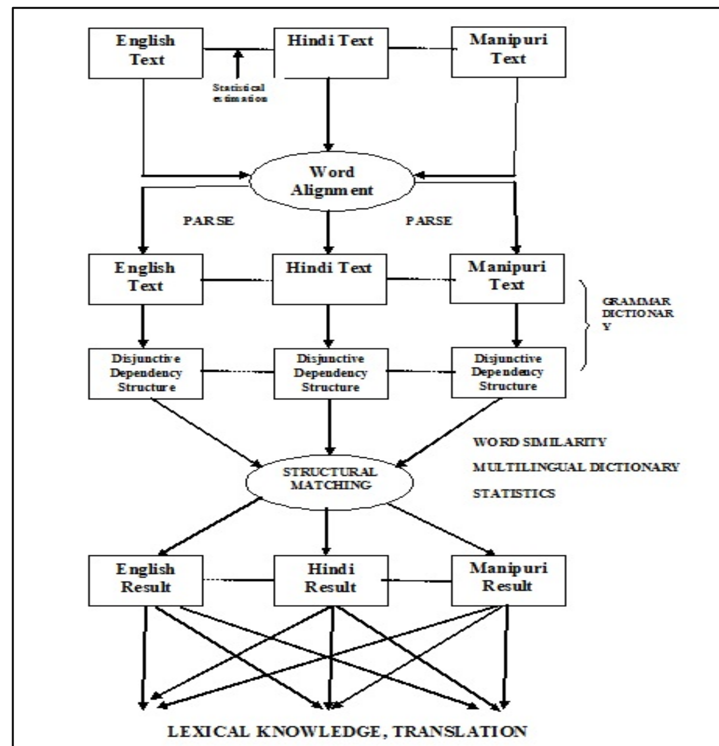


Figure 47: Parallel translation of Source language to Target language

5.9 Tools

This software has been developed using VB.NET as frontend, Microsoft SQL Server 2000 as backend and ASP.NET technology. The .NET languages, which includes Visual Basic .NET, C# and Jscript.NET or the Java-clone J# .NET the object-oriented and modernized successor to Visual Basic 6.0. Visual Basic.NET (VB.NET) is a redesigned language that improves on traditional Visual Basic, and even breaks compatibility with existing VB programs. C#, on the other hand, is an entirely new language. It resembles Java and C++ in syntax, but there is no direct migration path. C#, like VB.NET, is an elegant, modern language ideal for creating the next generation of business applications. Interestingly, C# and VB.NET are actually far more similar than Java and C# or VB 6 and VB.NET. Though the syntax is diferent, both use the .NET class library and are supported by the CLR. In fact, almost any block of C# code can be translated, line-by-line, into an equivalent block of VB.NET code. The occasional difference between these is C# support operator overloading while VB.NET does not.

1. ASP.Net

ASP.NET is a new and powerful server-side technology for creating dynamic web pages, the platform service that allowed us to program web applications and Web Service in any .NET language, with almost any feature from the .NET class library. When we request a page, the ASP.NET service runs (inside the CLR environments), executes our code, and creates a final HTML page to send to the client. ASP.NET offers several important advantages over previous Web development models. ASP.NET is a part of the .NET framework. As a programmer, we interact with it by using the appropriate types in the class library to write programs and design web forms. When a client requests a page, the ASP.NET service runs (inside the CLR environment), executes our code, and creates a final HTML page to send to the client.

2. VB.Net (Front-End)

VB.NET is the latest version of Visual Basic. It is a complete redesign that answers years of unmet complaints and extends the VB language into new territory. Some of the new features include.

- ✓ True object-oriented programming.
- ✓ Language refinements.
- ✓ Structured error handling.
- ✓ Strong typing.

3. SQL 2000 Server

SQL is both an easy-to-understand language and a comprehensive tool for managing data. It works with one specific types of database, called relational database.

4. Web Server (IIS)

Internet Information Service (IIS) Web server comes bundled with Windows 2000, Windows XP Professional, and Windows 2003 Server. IIS version 5.0 comes with Windows 2000, IIS version 5.1 with Windows XP Professional, and IIS version 6.0 with Windows 2003.

IIS plays an integral part in creating ASP.NET pages. Through its configuration, it directs Web requests to actual files on the system. In order to use ASP.NET, we must install IIS on our system. Just viewing the files locally on our hard drive won’t work because ASP.NET requires that the request go through IIS so that it can broker the ASP.NET request to the .NET Framework SDK.

5.10 Word Entry

Word entry or the microstructure of the MMD will contain the following list .

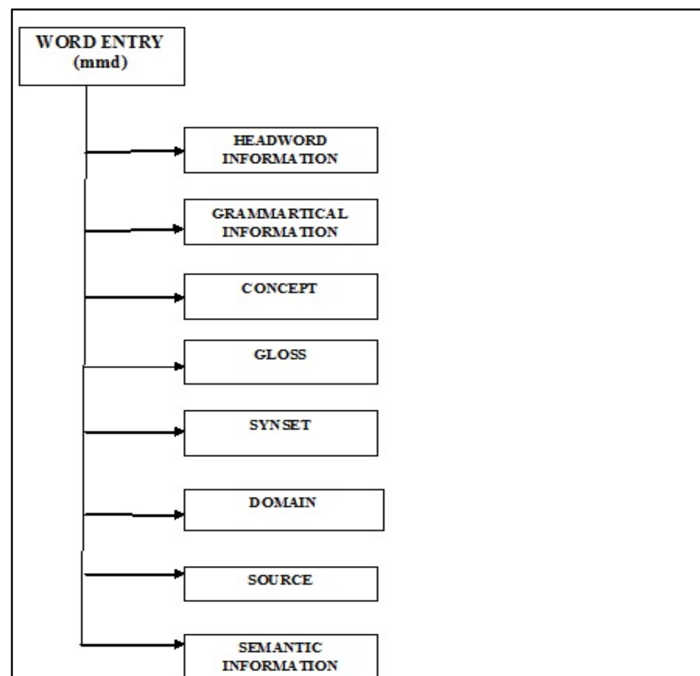


Figure 48: Word entry of MMD

5.10.1 Algorithm for Word Entry

Words are entered through the page by Administrator using the below algorithm after collecting from WordNet and Corpus.

(1) Enter the required lexical information for W*m*

(2) Creating an instance I*m*

if (!the mandatory information are contained) then,

go to stage (1) if(the synonym fields contain W*s*)

Find the word W*s* in the ontology

if (found)

retrieve the instance name, I*s*, of W*s*.

Saving the data and object properties of

I*m*, where, "Synonym" relation (object property) of I*m* is I*s*

else

Enter the required information to create the new instance, I*s*, for word W*s*

Saving the data and object properties of I*s* in the ontology

Saving the data and object properties of I*m*, where, "Synonym" relation (object property) of I*m* is I*s*.

Saving the "Synonym" relation for I*s* to I*m*.

Else Saving the data and object properties of I*m* in the ontology

Where, Wm = Manipuri Word (main), data type property of Im

Im = Instance (main)

Ws = Manipuri Word (synonym), data type property of Is

Is = Instance (synonym)

5.11 Word Class of English, Hindi and Manipuri

Table 22: Word Class in English

Sl.No.	CLASS/POS	DEFINITION
1	Noun	A noun is a word that identifies: a person a thing an idea, quality, or state
2	Verb	A verb describes what a person or thing does or what happens. For example, verbs describe
3	Adverb	An adverb is a word that's used to give information about a verb, adjective, or other adverb
4	Adjectives	An adjective is a word that describes a noun, giving extra information about it
5	Pronoun	A pronoun is a word or forms that substitute for a noun or noun phrase. It is a particular case of a pro-form.
6	Conjunction	Conjunction is the word that joins two words or sentences.
7	Preposition	A word governing, and usually preceding, a noun or pronoun and expressing a relation to another word or element in the clause
8	Determiner	A determiner is a word that introduces a noun, such as a/an, the, every, this, those, or many (as in a cat, the cat, this cat, those cats, every cat, many cats).
9	Exclamations	An exclamation (also called an interjection) is a word or phrase that expresses strong emotion, such as surprise, pleasure, or anger. Exclamations often stand on their own, and in writing they are usually followed by an exclamation mark rather than a full stop

Table 23:Word Class in Hindi

Sl	CLASS/POS	DEFINITION
1	Noun	वह शब्द जो किसी व्यक्ति, स्थान या वस्तु के नाम के रूप में प्रयोग किया जाता हैं उसे Noun कहते हैं.
2	Pronoun	सर्वनाम वह शब्द है जो Noun के स्थान पर प्रयोग किया जाता है
3	Verb	क्रिया वह शब्द या शब्दों का समूह है जो Noun के विषय में कुछ कहे अथवा उनके कोई कार्य का करना प्रकट करे.
4	Adjectives	Adjective (एडजेक्टिव) विशेषण वह शब्द है जो Noun का वर्णन करता है या उनके बारे में संकेत देता है, जैसे कि रंग-रूप, गुण-दोष, संख्या, आदि
5	Adverbs	क्रिया विशेषण वह शब्द है जो Verb, Adjective या अन्य Adverb के अर्थ को सुधारे या उसकी विशेषता प्रकट करे.
6	Preposition	सम्बन्ध सूचक-अव्यय वह शब्द है जो किसी Noun या Pronoun के साथ प्रयोग किया जाता है और जो उस Noun या Pronoun का सम्बन्ध किसी दूसरे Noun या Pronoun से प्रदर्शित करता है.
7	Conjunction	कंजंक्शन समुच्चय बोधक अव्यय वह शब्द है जो दो शब्दों या वाक्यों को जोडता है.
8	Interjection	इंटरजेक्शन विस्मयादिबोधक अव्यय वह शब्द है जो अचानक दिल से प्रकट होने वाली भावना को दर्शाता है

Table 24: Word Class in Manipuri

Sl	CLASS/POS	DEFINITION
1	Noun	[illegible]
2	Pronoun	[illegible]
3	Verb	[illegible]
4	Adverb	[illegible]
5	Adjectives	[illegible]
6	Postposition	[illegible]
7	Conjunction	[illegible]
8	Interjection	[illegible]

5.12 Aligning MMD and WordNet

As part of our quantitative comparison of computational multilingual dictionaries, we have created an alignment between two dictionaries at the level of lexical entries (i.e., a lexical alignment). That is to say, we align two lexical entries if, and only if, they share the same lemma and parts of speech. Obtaining a lexical alignment is exact and unambiguous, as it solely requires string matching. The encoded word senses typically vary in their number, granularity, domain-specificity, and conciseness of their definition. The WordNet synset {plant, works, industrial plant} defined as "buildings for carrying on industrial labor" describes, for instance, the same meaning as the fifth MMD word sense of plant defined as "a factory or other industrial or institutional building or facility". These two word senses should be aligned i.e., they should be part of a word sense alignment between the two dictionaries. Another MMD word sense of plant defined as "an organism that is not an animal [...]" denotes a different meaning and should therefore not be aligned to the WordNet synset.

Accordingly, a word sense alignment between two computational dictionaries d1 and d2 is a set of word sense pairs

ALLIGN(d1, d2) = {(sd1, sd2) | meaning (sd1) = meaning(sd2)}

Where s is the sense and d1, d2 are two dictionaries. Sharing the same meaning, the goal of an automatic word sense alignment method is thus to compare two word senses d1 and d2 and decide whether they are corresponding i.e. they share the same meaning. The pseudo code for aligning is given below in next section 5.12.1.

5.12.1 Pseudo Code of the Automatic Alignment Algorithm

function ALLIGN(WordNet, MMD)

alignment := ∅;

for each synset ∈ WordNet.getSynsets() do
// Step 1: Candidate extraction
candidates := ∅; for each word ∈ synset.getSynonyms() do

```
candidates ：= candidates ∪ MMD.getWordSenses(word);
// Step 2: Candidate alignment
for each candidate ∈ candidates do
simCOS ：=COS(synset,candidate);
simPPR ：=PPR(synset,candidate);
if simCOS ≥ τCOS ∧ simPPR ≥ τPPR then
alignment ：= alignment ∪ (synset, candidate);
 return alignment;
end.
```

5.13 Macrostructure and Microstructure of MMD

A macrostructure denotes the ordering of the dictionary articles. The index pages in MMD contain ordered lists of headwords linking to their corresponding article pages. Most printed dictionaries are ordered alphabetically. Macrostructure refers to the organization of the dictionary. The macrostructure can be divided into simple and complex, the former referring to the lexicographic macrostructure, which applies to only two-macrostructural components and the latter referring to three or more elements of the macrostructure, which every dictionary should ideally have. The macrostructural components of a dictionary may include:

- ✓ Table of contents.
- ✓ Preface.
- ✓ Users guide.
- ✓ List of abbreviations.
- ✓ Word list.
- ✓ Gead word list.
- ✓ Appendix.
- ✓ List of prefix/sufix.

The microstructure in dictionaries, the amount, content and structure of information following the dictionary entry are of interest. Microstructure may include various elements:

- ✓ Headword information.
- ✓ Part of speech.
- ✓ Pronunciation
- ✓ phonetic transcription like “ꯕꯣꯠ” of boat
- ✓ Tonal mark(for target language)
- ✓ Grammartical information.
- ✓ Concept.
- ✓ Gloss.
- ✓ Synset
- ✓ Antonyms, Hypernyms, Hyponyma.
- ✓ Morphological information.
- ✓ Semantic Information.
- ✓ Cross lingual information.
- ✓ Domain.
- ✓ Source.
- ✓ Semantic information.
- ✓ References.

5.14 Output Page of Word Entry

Figure 49: Word Entry Page

5.15 Trie Search

Based on the above data structure for storing stem words, a trie search program was developed which will give the address of the stem word input i.e. the stem pointer if the stem present.

A given input stem is searched character in the trie. Whenever a match occurs, with the character in the trie, the next character (alphabet) from the trie is extracted for comparison with the next character from the input stem.

If there is no match, the sibling address of the current characters is used to locate the next sibling unless the desired characters are found. If there is no sibling address then the stem is not present in the trie. When all the characters of the input stem word have been matched with the trie, the next character forming the trie must be a '*' symbol. If any other alphabetic character is found in the trie, then the stem is not available in the trie. The digit after the '*' constitute the stem pointer.

A different interface has been attached to the trie search program when the trie is used during the source language analysis. A surface level word will be used to find

out the stem and then the associated prefix and/or suffix along with the stem pointer with the help of the program. The details of this program are provided. The algorithm for the basic trie search program is:

1. Set the file pointer at the beginning of the trie dictionary.
2. For the input stem word to be searched in the trie dictionary, do
3. Read each character of the input stem and do step 4 until the end of the word is encountered.
4. For a character read from the input word, do
   (a) Read the character from the trie .
   (b) If they match, read the character from the trie until an alphabetic character is obtained. Set file pointer left by one unit. Go to step 4.
   (c) Else if they do not match, compare ASCII code of the two characters. If the ASCII code of the trie character is greater than the ASCII code of the input character is greater ASCII code of the input character then go to step 7. (Assuming that the two character code in the lower case only).

   Else if the ASCII code of the trie is lower than that of the input character, calculate the pointer address (for the sibling character) given in the trie by reading a character one by one. Read the character from the trie until the alphabetic or '*' character is encountered. Set the file pointer forward by sibling pointer address-2. Go to step 7.
5. Read the next alphabet from the trie which follow the '*' is encountered. If an alphabetic character is read before a '*' character, go to step 7.
6. Read the stem pointer from the tire which follow the '*' character. Print that the stem search is successful and print also the stem pointer. Go to step 8.
7. Print that the stem is not present in the trie dictionary.
8. Close the trie file and exit from the program.

5.16 Output Page of Search

Figure 50: Output page search by English word

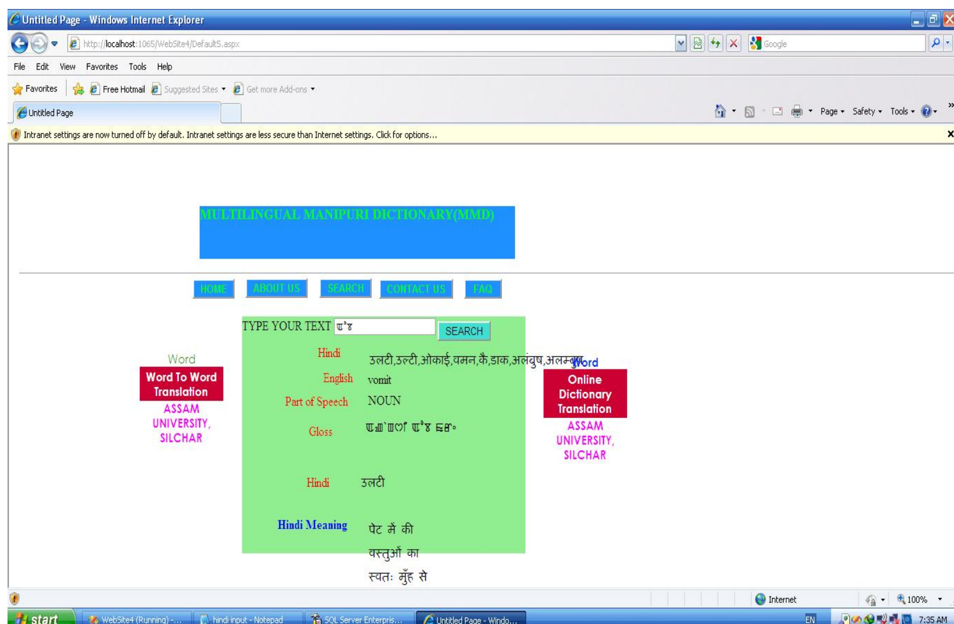


Figure 51: Search by Manipuri word

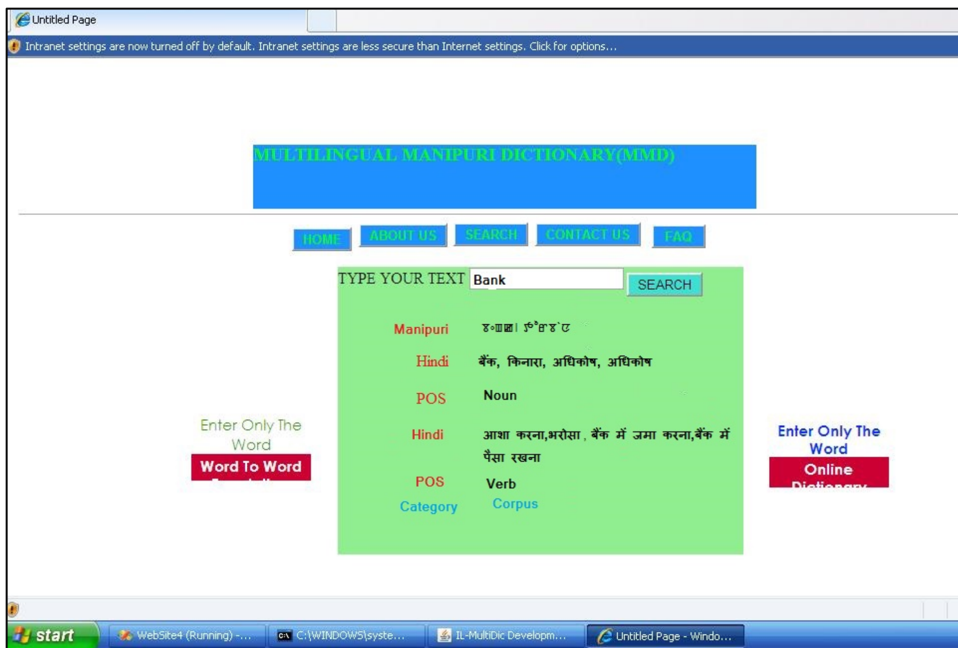


Figure 52: Search result by English for WSD

Figure 53: Search result by Manipuri For WSD

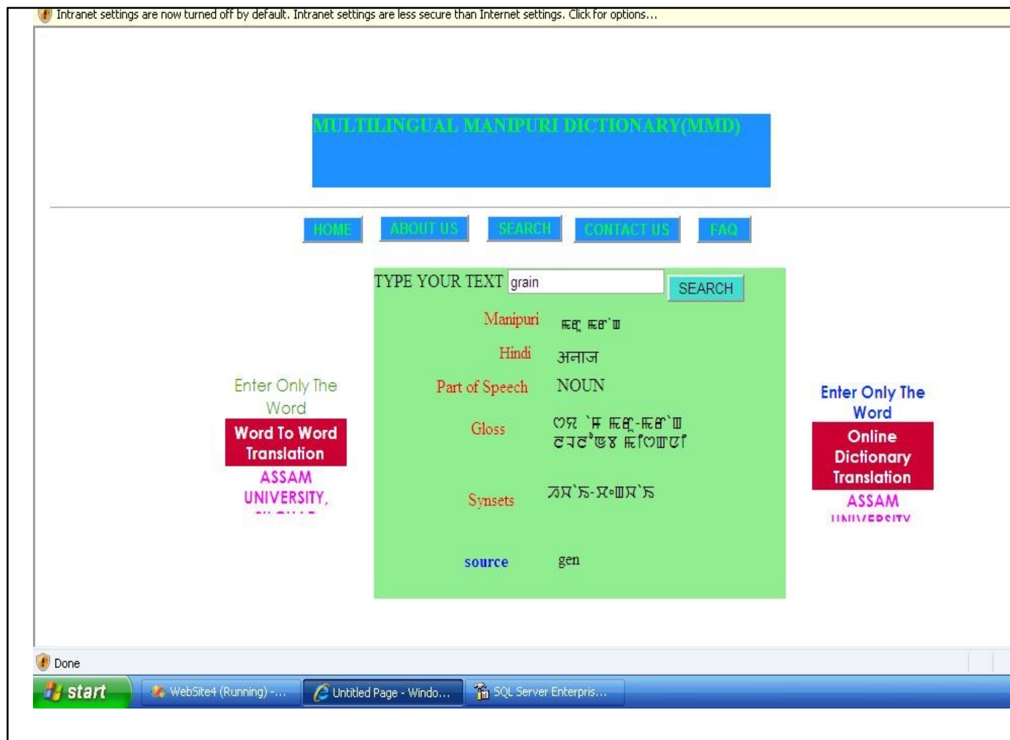


Figure 54: Search result by English to Manipuri

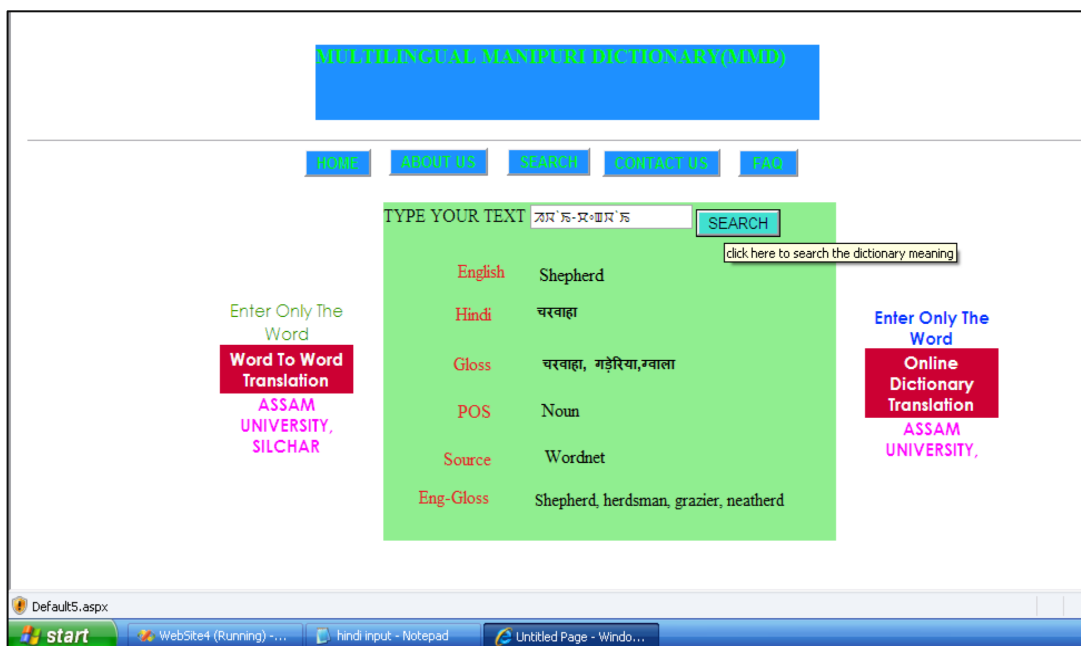


Figure 55: Search result by Manipuri

5.17 Statistical calculation for Word Entry in MMD

For statistical calculation of the total numbers of words entry it will go through the process for maximum number of word count per POS (Parts of speech).

Table 25: Total number of words calculated with WordNet

Sl. No.	**POS**	**Number of Word** (a)	**Number of Suffix (Manipuri)** (b)	**Total** (a X b)
1	NOUN	57,000	37	210900
2	ADJECTIVES	19,500	31	604500
3	VERBS	21,000	100	2100000
4	ADVERB	834	23	19183

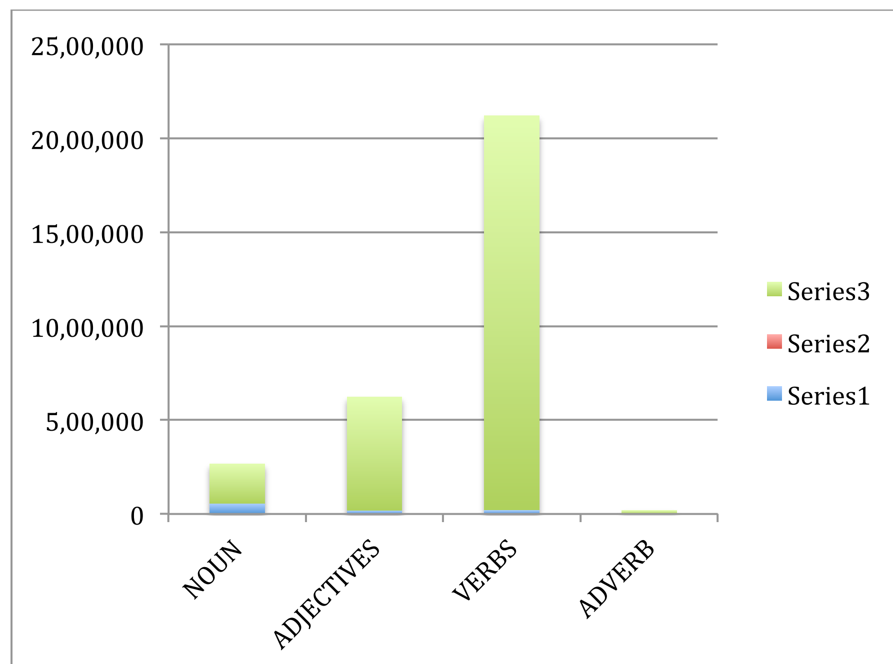


Figure 56: Graph showing maximum no. of word entry for MMD

5.18 Conclusion

The chapter has discussed about the design and searching of MMD with Manipuri, Hindi, English text translation. It can also use other Indian languages as target language, which will be very helpful in communicating without language barrier and moving up the knowledge chain. It will develop information processing tools to facilitate human machine interaction in Indian languages and to create and access multilingual knowledge resources/content. It will promote the use of information processing tools for language studies and research. It will also consolidate technologies, which developed for Indian language and integrate these to develop innovative user products and services. The sequential structure of dictionary, trie structure, trie search algorithm, advantages of tries over binary and hash table, complexity analysis of trie structure for dictionary, uses of suffix tables and its functionality, implementation of MMD, about tools and technology of MMD and word classes for the three languages used in MMD have been discussed. Lastly it discussed about algorithm for word entry and search algorithm and data calculation of MMD with the data from WordNet and corpus. Major initiatives will be knowledge resources, Knowledge tools, Translation Support System, Human Machine Interface System, Localization, and Standardization.