DESIGN OF AN SMS LEXICON FOR AN INDIAN LANGUAGE

A Project Report

submitted by

RAHUL CS

in partial fulfilment of the requirements

for the award of the degree of

MASTER OF TECHNOLOGY

under the guidance of

PROF. HEMA A MURTHY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

JUNE 2010

THESIS CERTIFICATE

This is to certify that, the thesis titled **'Design of an SMS Lexicon for an Indian Language'**, submitted by **Rahul CS**, to the Indian Institute of Technology Madras, for the award of the degree of **Master of Technology**, is a bona fide record of the research work done by her under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Place: Chennai-36

Date:

Prof. Hema A Murthy
Project Guide
Professor
Dept. of CS and Engineering
IIT Madras, 600 036

ABSTRACT

Mobiles are leading the list of commonly used devices. The most cost effective service provided by the device is **sms**. People are always in search for better service, and the service providers will obviously compete for providing better results. The objective of this project is to come up with a better approach for deploying **sms lexicon** for Indian Languages. Once implemented, it could be ported to any language platform, easily.

The objective is to improve user convenience in dealing with **sms** service, in the sense of reducing the number of key presses to type a message. The sample language we deal with is Hindi.

The fact that it is being deployed for Indian Language, makes it less comparable to other implementation techniques like **T9** [1]. To work out a solution, you will be provided with a sample lexicon.

The problem has been formalized as a task of coming up with concepts from data structures, and algorithms that achieve the primary objective along with satisfying the time and space constraints. We have started with a **Trie** [2] and optimized it as far as possible. We could keep on doing this, until some milestone being achieved, or some threshold is known to be reached.

The target platform is Symbian Operating System which is one among the best Mobile OS'. It provides good developer support too. So the work has focused on applying nice concepts and utilizing whatever tools available, to good effect. Once ported to the target platform, this tool could be used to send sms in Hindi.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

GUI	Gaphical User Interface
IDE	Integrated Development Environment
SDK	Software Development Kit
UI	User Interface
LUT	Look Up Table
KB	Kilo Byte
OS	Operating Systems
RAM	Random Access Memory

CHAPTER 1

Introduction

1.1 Overview

In the contemporary world, mobile phones play a vital role in our life. About 60.6 percentage of the total population of the world are mobile users. In India its about 49.6 percentage [3]. The device provides a wide variety of services like voice communication, entertainment like games, scheduler, calendar and calculator. Recent developments in this industry, like addition of internet facility makes them powerful enough to replace even PCs. Nowadays they support almost all the desktop applications. Added benefits like portability and compactness force us to replace land line phones and other wired voice communication services with this one.

Among the services provided by this device, the most appreciated service is obviously the Short Message Service(SMS). Statistics shows that on an average, an Indian sends 29 SMS per month [4]. Not just for communication, nowadays they provide many Value Added Services(VAS) like bill payments, train status enquiry and live cricket scores. The service providers are competing so hard to improve customer satisfaction and hence business.

Once the customer becomes satisfied with the services, the evaluation metric from his point of view will be the usability of the service, in turn the usability of device. So the service provider has to keep focus on that too. If you provide diverse services and if they are inconvenient to use for common people, they start showing reluctance to the Industry. Handling the device should be as simple as dealing with a calculator. Whenever possible, it should be hiding the complexity from the user. We could define this objective as *Device Transparency*.

1.2 Motivation

The section above describes how conscious the service providers are about the customer needs, in order to hold on in the market. They always keep on taking feedback about their technologies and they keep on updating the services based on the feedback. Most focused service will be the most appreciated one. In that sense, they have to put more focus on improving the Short Message Service(SMS), since Texting(synonym for SMS) is the widely used one among common people.

The process of sending a short message starts with the step known as message creation. This involves creating the message using symbols from the alphabet of that particular target language. As the size of alphabet increases, it becomes hard to map the alphabet symbols to the number keys on the keypad, since the domain of mapping is a constant figure. Whenever the alphabet size goes beyond the keypad size, we have to go for a many to one mapping. This makes the message creation step to be a cumbersome task, since the number of clicks made per character goes beyond one. As the alphabet size increases, the number of clicks to be made keeps on increasing by a factor of keypad size. Customers will always tend to reduce their load. They will be keen to reduce the number of clicks whenever possible.

This requirement introduces the concept of SMS lexicon. SMS lexicon together with word prediction fulfills the requirement of achieving reduction in button clicks and hence speeding up the process of message creation. One such technology which comes up with this facility for English language is known as T9(*Text on 9 keys*) predictive text technology. Since late 90s, most of the mobile developers have come up with this facility embedded in their devices.

In a developing country like India, people will prefer to communicate in their native language over English. In English, there exists an SMS word prediction scheme, which comes up with predictions about the words a user tend to type next, while he prepares a message. The system will be associated with a lexicon(*set of words from the vocabulary of a language*), which the system refers to, in order to make its predictions about the SMS words. So, it will be nice if we can come up with something similar to T9, for Indian languages too. The goal of this project is to achieve the above objective.

Nowadays, there is support for native languages in most of the mobile phones. Some of them have support for typing SMS in native languages. The task is to develop a module similar to T9, targeted for Indian languages.

1.3 Problem Definition - An SMS Lexicon for Indian Languages

Being motivated from the requirement mentioned above, we define the problem as developing a word prediction scheme for supporting Indian language SMS. We have to come up with a framework which satisfies all the constraints, and then deploy that on a selected mobile platform. The target language operated upon is Hindi.

1.3.1 An Operational Model

The framework will be having two components.

- SMS lexicon – Composed of selected legitimate Hindi words.
- Word prediction scheme – Come up with predictions about user intention based on his/her hints.

SMS Lexicon

A lexicon is a subset of a language's word set. The size of the lexicon (*the number of words in the lexicon*) depends on its domain of operation. Since the domain of operation under consideration is texting, the lexicon is limited in size. We are provided with a list of most frequently used SMS Hindi words, selected based on a statistical analysis. The objective is to come up with an appropriate data structure to deploy the lexicon so that it is compact and serves quick retrievability. In other words, an ADT (Abstact Data Type) with operation defined to be looking up a valid word.

The compactness of lexicon, in terms of technical terms, could be defined as the extent to which we can reduce the total memory usage for deployment of the lexicon. Since mobiles are low power and hence resource constrained devices, we are forced to ensure that the memory utilized is the bare minimum. In order to achieve this, we have to make the maximum use of the redundancy associated with the lexicon. The scheme should be focusing on eliminating redundancy.

Word Prediction

The user wants to reduce the number of button clicks as much as possible. But he should be making some clicks so that the system could come up with some predictions which meets the user needs. The user clicks are actually the hints about his/her intended word. These hints will be fed as input to the prediction system. The system has to use these hints to look up(operation defined on the ADT) the lexicon(ADT). The process should avoid taking too much time, in order to achieve the user transparency(*making sure that the user is not aware about the presence of complex underlying mechanism*). Although the response time of the system depends on the operational hardware platform, we should try to minimize the number of computational steps so as to make it compatible with the low end mobiles whenever possible.

1.3.2 A Target Platform

Once you are ready with a framework, it has to be deployed somewhere, in order to test it and to make a final decision about whether it suits for the purpose or not. Here the platform has been fixed to be Nokia S60 model mobile phones running on Symbian Operating System, which is well known for its specialization in smart phone technology.

1.4 Organization of the Thesis

The remaining part of the thesis has been organized as follows.

Chapter 2 briefly tells about the background study in this area and related work.

Chapter 3 talks about solution formulation.

Chapter 4 deals with issues related to to level design.

Chapter 5 explains the front end design and integration.

Chapter 6 explains implementation details.

Chapter 7 covers conclusions, limitations and future work.

CHAPTER 2

Background Knowledge and Related Works

This chapter covers the fundamental concepts that should be understood in order to deal with the problems at hand. This one also covers the related works and the findings which led to an admissible solution.

2.1 Dictionary Data Structures

There are a wide variety data structures to be selected from, in order to deploy a dictionary. Few of the candidates are BTrees,Hash Tables etc. Since the situation demands efficient space utilization along with sound retrievability, on choice narrows down to *Tries* data structure. Among them, we go for the one specialized for this purpose, which is Trie.

2.1.1 Trie

A **Trie** [2] is an n-ary tree, where n is called the arity of the tree. Arity is the upper threshold on the number of children any node in the trie can have, where each child in turn may be another trie(*sub trie*). An edge in the trie is an identifier which uniquely identifies one of the bindings between parents and children in the trie. At the top, there will be a single node without ancestors which is being known as the root of the trie. If we label an edge joining a parent and a child using a symbol say α , a set of size n (from which alpha being drawn) will be enough to recognize across the bindings between any parent and one its children. This labeling scheme in turn, uniquely identifies a path from root to one of its leaves as a sequence of symbols corresponding to those edges in the path. Suppose this set represents alphabet set of a particular language, then these paths represents possible words in that language. In other words, any word in the language can be represented by a unique path in the trie and a collection of words can

be represented by a unique trie. This is how trie happens to be a sound candidate data structure for representing lexicon of a language.

Figure 4.3 shows trie representation of the lexicon of a language whose alphabet set is $\{\alpha, \beta, \gamma, \delta\}$.

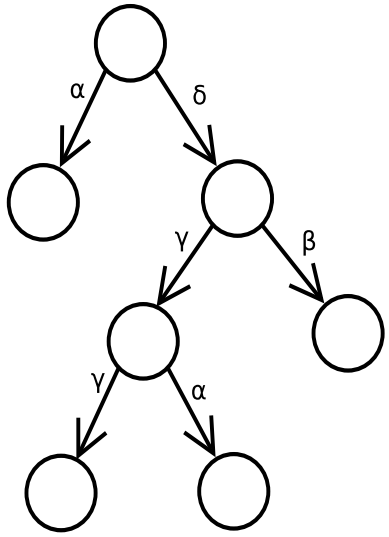

Fig. 2.1: A trie representing a sample lexicon

The set of words covered by this trie are:

- α
- $\delta\gamma$
- $\delta\gamma\gamma$
- $\delta\gamma\alpha\beta$

In this representation, it could be noted that common prefixes are being replaced by single occurrences. This is how it achieves reduction in storage space and the amount achieved is directly proportional to the common prefix quantity associated with the word set.

2.1.2 Patterns based Trie

In simple trie, the property that the domain space for selecting branch names could be an Alphabet of a language, makes it a suitable representation for the lexicon of that

particular language. This property will be retained even if we evolve the alphabet set by adding some more elements to it. If the newly added elements are combinations of symbols already there, then they may also take up the job of representing sub patterns in the trie, as the Alphabet symbols do. The benefit achieved because of these sub patterns depends on the frequency of their occurrences in the target word set. Thus the domain space for branch names also depends on the target word set, rather than just on the language alphabet set as in a simple trie. So the new form of trie has an associated **look up table**. Trie structure will be edge labeled with numbers which will serve as indexes in to the look up table(See figure 2.2). The selection of candidates for look up table plays a big role in the efficiency of these Tries.

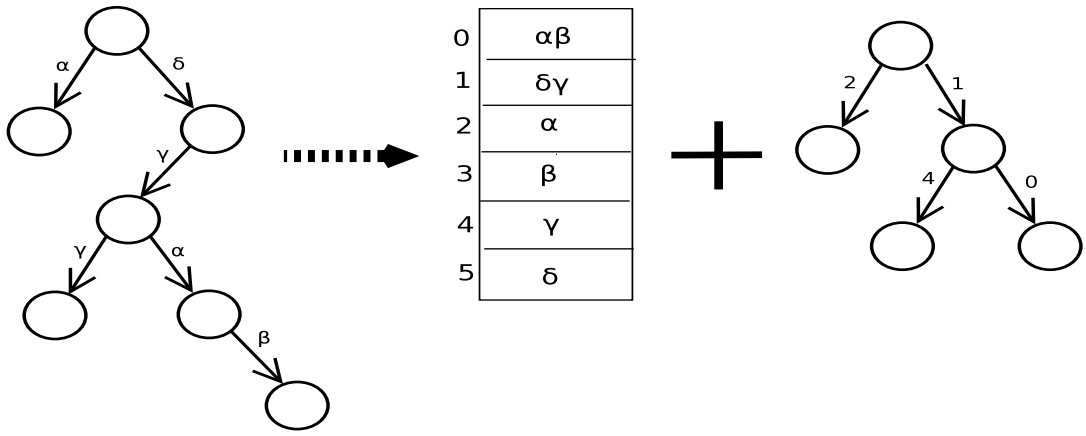

Fig. 2.2: Patterns based Trie - An extended version of normal trie

2.2 A Pointerless BDD Package

Binary Decision Diagrams(BDD) play a vital role in modern digital circuit design. There are BDD packages which are supposed to facilitate the manipulation of BDDs. Binary Decision diagrams are basically tries which represents a binary language word set. As the applications demand, BDD packages should be associated with functionality that let the BDD to grow dynamically. Basically this requirement brings the pointer based implementation in to picture.

But a new design of BDDs [5] in which they fulfill the requirement for pointers with statically allocated memory, as long the upper bound on the size of the BDD is known. There will be a fixed amount of statically allocated memory. Whenever an application requests for memory, a portion this static memory will be allotted for it. If it is pointer in dynamic allocation, this scenario uses array index as the reference parameter. This provides faster access(random access) and possibly storage space saving along with garbage collection and other pointer based services.

The prerequisite is the knowledge about the maximum storage space requirement. For applications like trie based lexicon implementation, where we could get a clear idea about maximum storage requirement, we could borrow this pointerless concept to tries, and could make maximum use of it.

2.3 Program Development on Mobile Devices

Mobile devices are very similar to PCs in Architecture. They also do have computational elements and memory elements. They are also governed by operating systems as is the case with PCs. This makes dealing with the device easier for someone who are familiar with PCs. The class of people dealing with the device could be mainly divided in to mobile users and mobile application developers. While the mobile users enjoys mobile applications, developers focus on providing improved mobile applications and hence promote mobile users. The process of mobile application development does not vary much from PC application development.

2.3.1 Mobile Operating System

As Linux or Windows operates on a desktop PC or laptop,a Mobile operating system interfaces users(user applications) to hardware mobile devices. Since the target devices are resource conscious, the designers of mobile OS focus on that too, rather than just targeting ultimate output to the user. There are many mobile OS, but one of the most prominent one among them is Symbian OS. It holds 50.3% of the total mobile market [6] and is being widely accepted. Now it is open source too. It provides almost all the

operating system services as a normal Operating System does for a PC.

Keeping the target platform in mind the designers of Symbian OS has followed principles like,

- Resources associated with the target platform are scarce.
- User time is precious.

The Symbian Operating System

Symbian is a multitasking multithreaded operating system. It ensures efficient memory utilization by the use of what are called Dynamic Link Libraries(DLLs). DLLs ensures on demand loading of kernel services, and at a time only one instance of a service will be residing in memory. Also this is an event based operating system. Any change in device's state could be identified by generation of an event. Then its about handling that event representing that change. Kernel will be ready to handle most of the events by use of some default handlers. User applications may or may not handle them depending on whether they are prioritized enough to do it or not. Several classes of events are there with different levels of access permissions. Once a developer get access to one of the levels, he can decide upon handling all the events that comes under that particular level and everything below that.

Development could be done in object oriented programing language called Symbian c++ [7], which is being well tuned to match the platform. The operating system and application software follow an object oriented design called Model-View-Controller(MVC). Development in Symbian OS Applications in Symbian will be having three components.

- Model - This component is defined in the form of document class which is available as part of symbian API. This has to deal with allocations and deallocation of memory.

- View - This one deals with GUI and hence it decide upon how to present the application to the user.

- Control - This is the component which decides the behavior of the application. It contains the definition of all the event handlers that the application wants to deal with.

2.3.2 Mobile Emulators and Mobile Application IDEs

Most of the industrial giants have come up with emulators that emulates the target device on a PC in order to experiment with mobile applications. They also provides IDEs, that facilitates mobile application developer's job. The IDEs generate executables for both emulator and mobile device. Even it is possible to perform on device(mobile) debugging. It is also being ensured that the mobile application development languages do not vary much from programming languages so as to ensure that it won't be hectic for someone coming from the world of PC to the mobile development environment.

CHAPTER 3

Proposed System

This chapter explains how we approach the problem. First we look at the task to be dealt with, as a whole. Then the break down the job to be done. Handle individual modules separately, built them up and integrate them to produce the target system.

3.1 Problem

The objective is to develop an SMS lexicon for Indian Languages. We are provided with the target platform specification, and word set. The target platform is a mobile device. The experimental word set is in Hindi. What we have to do is to develop an application that resides in the device along with the input word set. Whenever user wants to send a message in Hindi, this tool should play a supporting role such that it comes up with predictions about the word intended by the user. As soon as the user starts typing, it should capture the patterns and access the word set for predicting the word it has to come up with. Since the keypad to alphabet set, follows a one to many mapping, a combination of key press will result in multiple possible combinations of alphabet symbols that the tool has to deal with. With the word set in back of the mind, it has to classify the list of patterns to, those which are valid prefixes and those that are not. Then it has to come up with a mechanism that lets the user to select one among those valid patterns.

3.1.1 Design Challenges

Since the target platform is a mobile device, the tool should be demanding only a few resources. The resource set can be classified into memory and computational requirements. The performance could be measured as difference between the time user performs an action (like a key press) and the time the user gets a response from the tool

(some state change in display). Here we are provided with an upper bound on the expected response time of the tool. So it is just about bringing the computation time down to within this bound rather than focusing on optimizing it.

The real matter of worry is memory. The memory components of the target device are power sensitive in nature. We are forced to reduce the memory usage as much as possible. So we have to keep on optimizing this parameter whenever possible. The memory occupied includes both code memory and data memory. The design should come up with a representation of the lexicon that occupies minimum amount of memory and the parser module should also be well optimized to occupy minimum memory, along with satisfying the response time constraint, mentioned as above.

3.2 Approaching the Task in Hand

One of the most widely used strategies to solve a problem, is by relating it to solved problems already there around us. We also follow the same habit. There are a number of lexicon based problems around, only differ in the domain of application. If we look a bit deep in to those, it could be noted that most of them use trie as such, or some variation of the same, which suits their purpose.

It does not seem to be a bad idea to start working out the task keeping trie as the base data structure. Later we could go on evolving, whenever possible and effectively.

3.2.1 Deploying the Lexicon

In a trie, each node is an alphabet, contributing somehow to the overall construction of the lexicon. Making the nodes to be made of combinations of alphabets rather than single ones, so that a possible sequence of nodes could be replaced by a single node. This will introduce a need for defining a new *set of domain constructs*(sort of book keeping) from which, the node values for the data structure will be drawn.

3.2.2 Look Up Table

In order to deploy the lexicon in an effective manner, we first modify the language Alphabet itself, so as to make the word to be made of bigger constructs and hence lesser number of constructs. Now if we use these constructs to represent a word in some data structure, then the count of data structure units used to represent these constructs should also come down. Now its about storing this modified set of constructs somewhere and referring to this book keeping information from the storage structure. Each entry in the data structure will be referring to the table.

3.2.3 A Storage Structure for Lexicon

We make use of the look up table by storing the lexicon as a collection of references from the trie to the table as shown in figure 2.2. The path in the trie represents a sequence of references into the look up table, which in turn corresponds to a word. The data structure used here is a modified form of trie. As already mentioned, the initial modification we make is replacing the basic alphabet set with an improved collection of constructs drawn from the, well built look up table. Next we try altering the trie in a peculiar way which intuitively seems to be effective for our purpose, as explained in sections 4.2 to 4.4

3.3 Dealing with Front End

Once the lexicon is guaranteed to meet the criteria, it is about using it to meet user requirements. The user will be producing input in the form of key presses. The system has to read it and process it based on the reference data stored in the form of lexicon, that has been provided as part of the requirements(the lexicon).

Identifying the valid alphabet combination corresponding to a key combination involves parsing the data book (the stored form of lexicon) with possible candidate alphabet sequences, and identifying whether they succeed in parsing the lexicon to prove their validity.

So the front end should be able to,

- Ensure availability of the lexicon
- Provide a module to mapping from valid button combination to possible alphabet combinations.
- Filter out the valid patterns based on the dictionary
- Reflect the output back to the user, with provision to select among different alphabet sequences.

3.4 Conclusion

The whole task is broken down in to,

- Setting up an efficient foundation for the tool to facilitate the way it deals with the user.
- Deciding upon how the tool should be made to respond dynamically to the user events, relying on the foundation already built.

The next two chapters cover the suggested solution in detail.

CHAPTER 4

Low Level System Design

This chapter explains the underlying concepts in detail. We develop a mechanism on which the actual system rely heavily for its smooth operation. First of all we convert the lexicon in to a form which conforms to the requirement of faster accessibility and reduction in storage space. Then this module could be used by the front end for satisfying user needs at a higher level.

4.1 Look Up Table

Words in a language are made of symbols from its alphabet set. Look up table is where the modified alphabet set for the language resides. It will be a super set of the original alphabet set for the language. The additional elements are frequently combinations of symbols from the actual alphabet set, so that they could also be used for the construction of words. The added patterns play a big role in the overall performance of the system. The selection of patterns for the look up table should be such that the total number of look ups we perform to construct the lexicon is minimum.This could be achieved by, populating the look up table with possibly lengthy frequently used patterns and increasing the number of patterns. Both strategies will result in increase in look up table size. But we can't let the table to grow without control since this is again consumption of memory.

So the task in hand is,

- to fix the look up table size.
- select the patterns to populate the look up table.

Both have to be dealt with extreme care.

In order to fix the look up table size, it should be noted that the overall size of the table is the product of the number of entries and the size of the largest pattern. Now the job is being broken down to fixing the maximum pattern size, and fixing the look up table size.

Patterns are the construction units of each word in a word set. For constructing words, we either use the entire pattern or is not used at all. So, it is fair to assume that the pattern length should be decided based on the length of words present in the word set. First we calculate the average length of a word(μ) from the word set. Calculate standard deviation(σ). The difference $\mu - \sigma$ is fixed to be the rough estimate of the minimum length of word present in the word set. We fix this parameter as the maximum pattern length. i.e this will be the length of the longest pattern present in the look up table.

Next is to fix the number of entries present in the look up table. The system will be using a sort of indirect addressing to refer to the table elements. Because of this, it is being preferred to fix the number of entries such that the address of reference(in this context the address is index to the table) will be in byte boundaries. We confine to this to make sure that the storage and retrieval of reference addresses becomes simple and effective. Therefore we have fixed the number of entries as 256, as the original word set itself does not exceed 7000 words.

Populating the look up table

Once the table size got fixed, we have to fix the candidate patterns for the table. This selection should satisfy two criteria

- It should be possible to construct each word in the lexicon using patterns in the look up table.
- On average, the number of reference made to the table for the purpose of constructing the word should be minimal.

The first criteria could be satisfied just by making sure that the table covers the alphabet set for the language. Next, we define the **benefit factor** for a pattern as,

benefit factor=length of pattern $\times$ frequency of occurrence

The benefit availed is the space that can be saved by storing lengthy and frequently used patterns. We will set, a minimum threshold that the benefit factor should cross, as the criteria for the pattern to satisfy to get in to the look up table. The threshold should be fixed such that the number of patterns satisfying this criteria is just enough to fit the table. We start the algorithm by fixing this factor to a reasonable value based on intuitions. Later we tune this, till all the criteria are met.

For setting up the look up table, we use Algorithm 1. The execution sequence is explained with a sample lexicon of three words.

{incarnation,caption,vision.}

Let the maximum pattern length parameter be 3. Let the look up table size be 15. Let the minimum benefit factor be 2.

The main procedure identifies the most beneficial pattern with the help of **Select-Pattern()** (refer function SelectPattern() in Algorithm 1)function. The function selects a pattern that produce the maximum benefit.

For example, in the given list of words, the pattern that produce maximum benefit during the first call to function **SelectPattern()** is, *ion*. In this pattern, the pattern length is 3. The frequency of occurrence of this pattern in total is 3, once in each word. Now the benefit achieved of this pattern is (3-1) $\times$ 3. This benefit factor 6, crosses the minimum threshold and will get selected. Look up table will be updated with the newly selected pattern. Later call the **RegenerateDictionary()** function in order to chop off all the occurrences of that pattern from the dictionary(stored as an array of words). This is to imply that a new candidate pattern has got added to the look up table in order to cover all the chopped off instances in the dictionary. After a call to **RegenerateDictionary()** the lexicon becomes, { incarnat,capt,vis }. We thus repeat **SelectPattern()** and **RegenerateDictionary()** in each iteration till, either the lexicon becomes empty or look up table get fully populated.

Algorithm 1 Generating appropriate patterns to set up the look up table.

1: **procedure** GeneratePatterns
Require: Original Dictionary file
Ensure: LookUpTable.
2: $LookUpTable \leftarrow alphabetset$
3: $patterncount \leftarrow 0$
4: $currentPatternLength \leftarrow 0$
5: $currentPattern \leftarrow null$
6: $maxBenefit \leftarrow initmax$
7: $maxPatternLength \leftarrow initMaxPatternLength$
8: $patternPresentFlag \leftarrow 1$
9: **for** $i =$maxPatternLength **to** 1 **step** 1 **do**
10: **while** $patternPresentFlag = 1$ **do**
11: $patternLength \leftarrow i$
12: **if** $patternLength = 1$ **then**
13: $maxBenefit \leftarrow 1$
14: $patternBenefit \leftarrow 1$
15: **else**
$patternBenefit \leftarrow patternLength - 1$
16: **end if**
17: **if** $dictionary = empty$ **then**
$return$
18: **end if**
19: $currentPattern \leftarrow SelectPattern(patternLength, maxBenefit)$
20: **if** $currentPattern \neq null$ **then**
21: $patternPresentFlag \leftarrow 1$
22: $RegenerateDictionary(currentPattern)$
23: $LookUpTable[patternCount] \leftarrow currentPattern$
24: $patternCount \leftarrow paternCount + 1$
25: **else**
26: $patternPresentFlag \leftarrow 0$
27: **end if**
28: **end while**
29: **end for**
30: **end procedure**

```
1: function SelectPattern(patternLength,maxBenefit)
2: currentMaxBenefit ← maxBenefit
3: for i =1 to wordCount step 1 do
4:    tempWord ← Dictionary[i]
5:    for j =1 to length(tempWord)-patternLength step 1 do
6:       pattern ← tempPattern[j...j + patternLength]
7:       frequency ← CountFrequencyInDictionary(pattern)
8:       benefit ← (length(pattern) − 1) × frequency
9:    end for
10:   if benefit > currentMaxBenefit then
11:      if pattern not in LookUpTable then
12:         currentPattern ← pattern
13:         return
14:      end if
15:   end if
16: end for
17: end function
18: function CountFrequencyInDictionary(pattern)
19: freq ← 0
20: tempPattern ← null
21: for i =1 to wordCount step  do
22:   load dictionary
23:   tempWord ← Dictionary[i]
24:   for j =1 to length(tempWord)-length(pattern) step 1 do
25:      tempPattern ← tempWord[j...j + length(tempWord)]
26:      if pattern =tempPattern then
27:         freq ← freq + 1
28:      end if
29:   end for
30: end for
31: returnfreq
32: end function
```

```
1: function RegenerateDictionary(pattern)
2: load dictionary
3: for i = 1 to WordCount step 1 do
4:    prevIndex ← 1;
5:    tempWord ← Dictionary[i]
6:    for j = 1 to (length(tempWord)-length(pattern) step 1 do
7:       if pattern[1...length(pattern)] = pattern[j...(j + length(pattern))] then
8:          tempPart ← tempWord[prevIndex...(j − 1)]
9:          write back tempPart to Original Dictionary File
10:         j ← j + length(pattern) − 1
11:         prevIndex ← j
12:      end if
13:   end for
14: end for
15: returnfreq
16: end function
```

The changes happens to the data at each iteration is explained in the table 6.1.

Iterations	Look up table	Dictionary of patterns	Benefit of current pattern
0	{}	{incarnation,caption,vision}	-
1	{ion}	{incarnat,capt,vis}	6
2	{ion,ca}	{in,rnat,pt,vis}	2
3	{ion,ca,i}	{n,rnat,pt,v,s}	1
4	{ion,ca,i,n}	{r,at,pt,v,s}	1
5	{ion,ca,i,n,t}	{r,a,p,v,s}	1
6	{ion,ca,i,n,t,r}	{a,p,v,s}	1
7	{ion,ca,i,n,t,r,a}	{p,v,s}	1
8	{ion,ca,i,n,t,r,a,p}	{v,s}	1
9	{ion,ca,i,n,t,r,a,p,v}	{s}	1
10	{ion,ca,i,n,t,r,a,p,v,s}	{}	1

Table 4.1: Execution sequence of the procedure GeneratePattern

4.2 A Storage Structure for the Lexicon

As already mentioned, we start the work with a trie, and a modified alphabet set(LUT). We use the same strategy of inserting the word to the simple trie. We identify the right construct(pattern) from the look up table. For that we look in the LUT for the pattern that is the longest prefix of the word under consideration. Once we found it, we have

to add the index of that pattern at the right position in the trie, representing the position of that pattern in the word. The insertion of pattern indices, start from root. We check for an edge leaving the root, labeled with this particular index under consideration. If it is not there, we add it. That edge lead us to a new node. This new node and the remaining suffix of that word to be added to the trie, form the new input for the procedure. Consider the node as root of a trie, and part of the word left, as a word to be inserted. Repeat the above step we did with actual root and actual word, for this new input. As we progress, the length of word in hand will reduce. We repeat till the whole word get added to the trie. Now the whole procedure has to be repeated for all the words in the lexicon.

The following example (Figure 4.1)shows a list of Hindi words, its LUT and the corresponding trie.

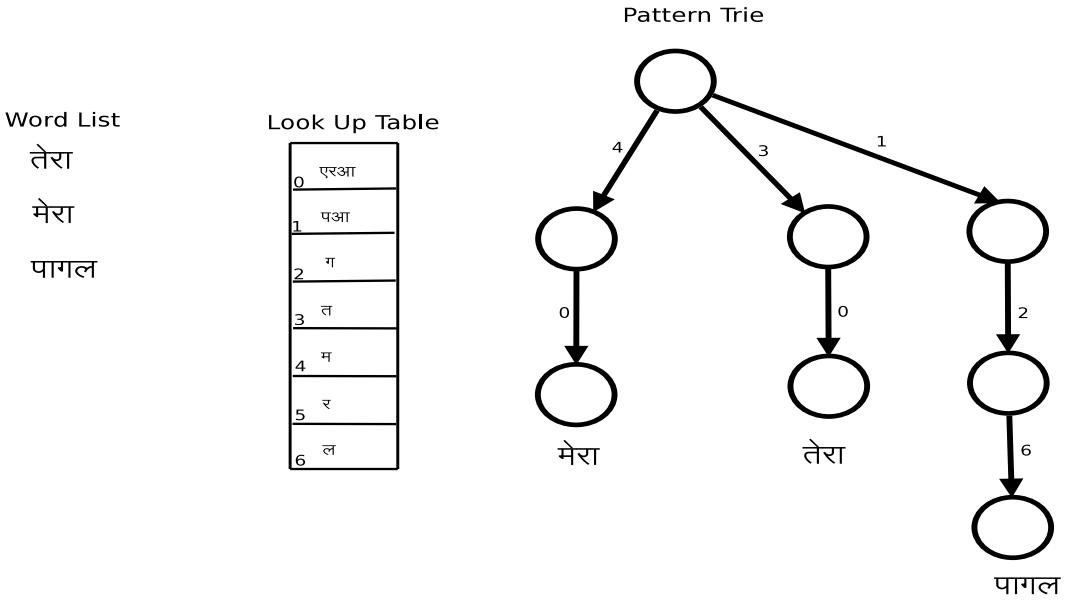

Fig. 4.1: A simple pattern trie

This is the algorithm designed for trie construction. Each node in the trie(Figure 4.2) has three fields.

- Pattern Index
- Side Index
- Down Index

- validFlag

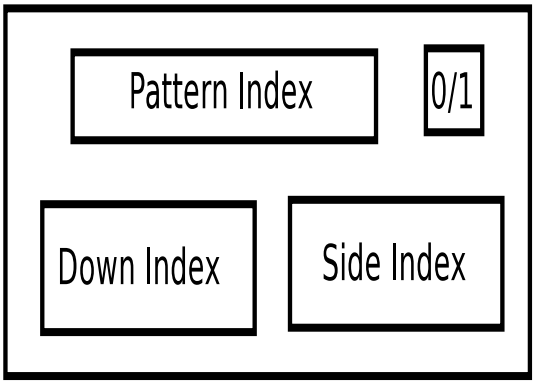

Fig. 4.2: model of nodes in the structure

Also we use the strategy of replacing the pointers with indices to statically allocated memory. We declare the amount of static memory allocated as an upper bound on the total need. Once we perform a test run, will get the exact amount of memory needed. Then we could fix the memory as per our need.

Among the node fields defined above, **lookUpTableIndex** points to the entry in the LookupTable to which the current node bound to. **sidePointer** field holds the index of the slot allotted to the node next to the current node in the same level. **downPointer** contains the index of the slot allotted to the first child of the current node. The **validFlag** bit identifies whether a prefix of a valid itself is valid or not. A variable **nodeCount** points to the next free slot, initialized to 1. Zeroth slot is assumed to be occupied by the root of the trie. The statically allocated memory is addressed with the name, **nodeList**. **patternCount** represents number of patterns present in the look up table.

At the end of the process, each unique paths in the trie, will represent one of the target target words.

4.3 Optimizing the Data Structure

In a normal trie, we eliminate redundant prefixes. It will be really effective, if we could eliminate redundant suffixes too. For that we traverse the trie, find out the replicated

Algorithm 2 Constructing the pattern based Trie

1: **procedure** CreateTrie
Require: Original Dictionary file, LookUpTable
Ensure: PatternBasedTrie
2: **for** i =1 **to** wordCount **step** 1 **do**
3: $tempWord \leftarrow Dictionary[i]$
4: $tempPointer \leftarrow nodeList[0]$
5: **for** j =1 **to** length(tempWord) **step** 1 **do**
6: $patternIndex \leftarrow FindPrefix(tempWord, j, length(tempWord))$
7: $pattern \leftarrow LookUpTable[patternIndex]$
8: $patternLength \leftarrow length(pattern)$
9: $tempPtr \leftarrow nodeList[tempPointer[down]]$
10: $presentFlag \leftarrow 0$
11: **if** $tempPtr \neq null$ **then**
12: **while** $tempPtr[side] \neq end$ **do**
13: **if** $tempPtr[lookUpTablleIndex] = index$ **then**
14: $presentFlag \leftarrow 1$
15: break the loop.
16: **end if**
17: $tempPtr \leftarrow nodeList[tempPtr[side]]$
18: **end while**
19: **end if**
20: **if** $presentFlag \neq 1$ **then**
21: $nodeCount \leftarrow nodeCount + 1$
22: $newNode \leftarrow nodeList[nodeCount]$
23: $newNode[lookUpTableIndex] \leftarrow index$
24: **if** $tempPtr = null$ **then**
25: $tempPointer[down] \leftarrow nodeCount$
26: **else**
27: $tempPtr[side] \leftarrow nodeCount$
28: **end if**
29: $tempPtr \leftarrow nodeList[nodeCount]$
30: **end if**
31: $tempPointer \leftarrow tempPtr$
32: **end for**
33: **end for**
34: **end procedure**
35: **function** FindPrefix(word,index,length)
36: **for** $i = index$**to** $length$**step** 1 **do**
37: **for** $i = j$**to** $patternCount$**step** 1 **do**
38: $pattern \leftarrow LookupTable[j]$
39: **if** $word[i...(i + length)] = pattern$ **then**
40: $return(j)$
41: **end if**
42: **end for**
43: **end for**
44: **end function**

tails,eliminate one of them and adjust one of the parents of those tails, to make it point to the common tail(shown later in section).

In order to locate the common tails, we exploit the property that these tails are suffixes of some words, and whenever there is a common suffix for a pair of words, their reversed form will be having a common prefix. Based on this analogy, we reverse all the words in the lexicon and form a new reversed trie from this reversed set of words. Now, it is easy to locate the words with common prefix in the new formed list since we just have to perform a traversal in the corresponding trie. If we perform an inorder traversal on the reversed trie, and list the words in the sequence they got visited, two adjacent ones in the sequence represents the possibility of having a suffix in common, in the original trie. Once we get a pair like this, we go back to the actual trie, in order to look for the common suffix. Then we traverse down, the path representing both the words, in order to merge their suffix in common.

First, create the reversed list of words, in terms of patterns from the look up table. Then the algorithm 2 mentioned above, with input as the new list, could be applied to create the reversed trie. The example below in Figure 4.3 shows how the optimization applicable to the trie given above can be performed.

Now the Algorithm 3 performs inorder traversal on the new trie in conjunction with applying suffix factoring on the original trie. The algorithm uses a **nodeStack**(where nodes will be held temporarily) to perform inorder traversal. The list of nodes for the reversed trie is given by **revNodeList**.

4.4 Recovering Unused Memory

Now eliminated redundant tails cause holes in the statically allotted memory. For that we do some compaction. This is being done by moving the nodes from the end of the list to fill the holes in between, so that at the end of the process, all the holes get accumulated at the end of the nodeList. We could write the sequence of valid nodes to a file. Now we know the amount of static memory required exactly. Allocate that much static memory and load the trie back from the file. This will provide the lexicon loaded

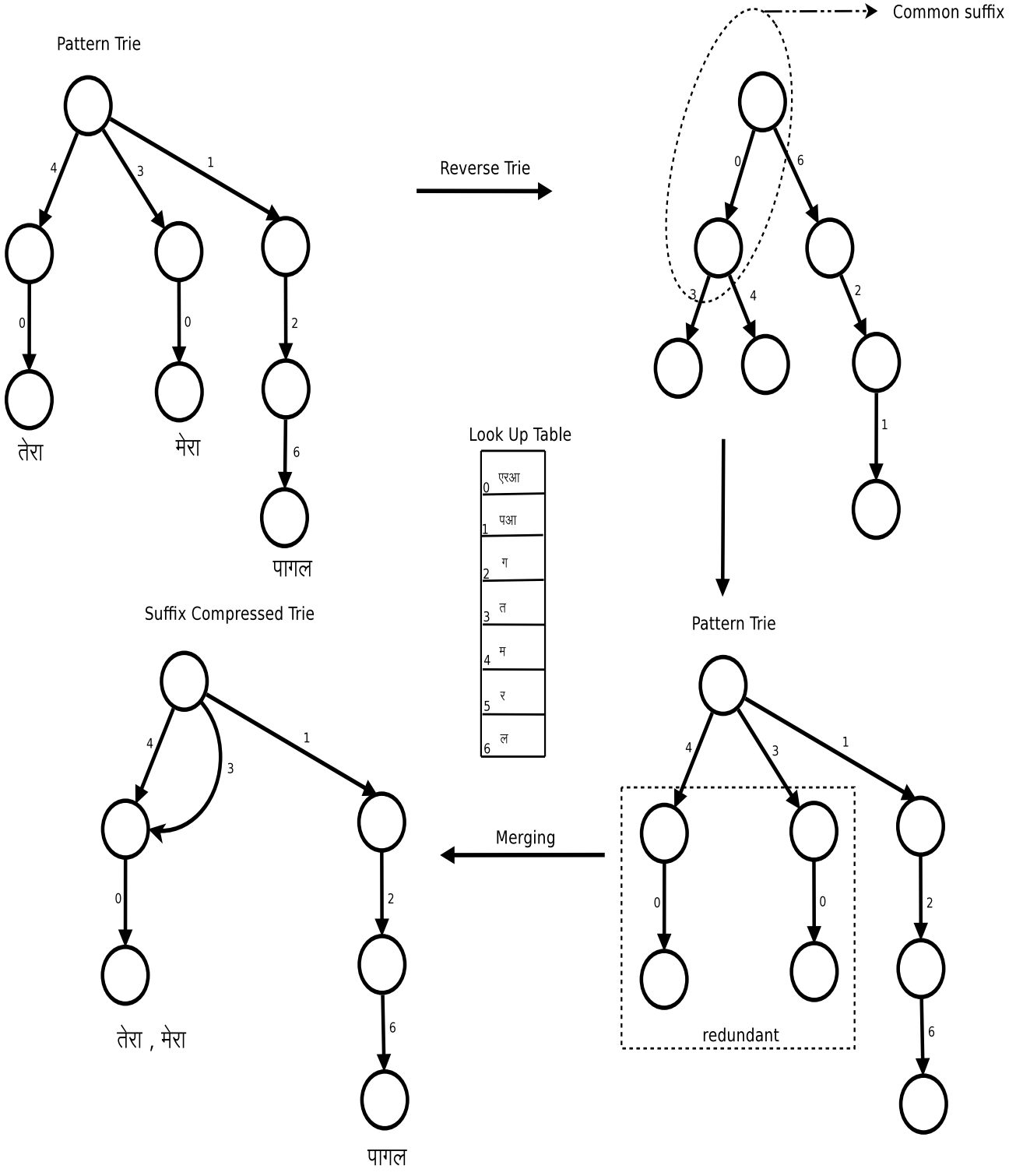

Fig. 4.3: Way to final storage structure represention

Algorithm 3 Algorithm for parsing and suffix compression

1: **procedure** CreateSuffixTrie
Require: Reversed Dictionary file, LookUpTable, Original trie
Ensure: PatternBasedTrie

2: $end \leftarrow 0$
3: $null \leftarrow 0$
4: $buffBackUp \leftarrow null$
5: $buff \leftarrow null$
6: $PUSH(revNodeList[0])$
7: **while** $nodeStack \neq empty$ **do**
8: $temp \leftarrow POP()$
9: **while** $temp \neq null$ **do**
10: **if** $IsWordBoundary(temp)$ **then**
11: **if** $buffBackUp \neq null$ **then**
12: $commonPrefixLength \leftarrow FindCommonPrefixLength(buffBackUp, buff)$
13: **if** $commonPrefixLength \neq 0$ **then**
14: $TryCompress(buff, buffBackUp, commonPrefixLength)$
15: **else**
16: $buffBackUp \leftarrow null$
17: **end if**
18: **end if**
19: $buffBackUp \leftarrow buff$
20: **end if**
21: $tempDown \leftarrow revNodeList[temp[down]]$
22: **if** $AlreadyVisited(temp) \neq false$ **then**
23: $tempDown \leftarrow tempDown[side]$
24: **if** $tempDown[side] = null$ **then**
25: $tempDown \leftarrow null$
26: **else**
27: $tempDown \leftarrow revNodeList[tempDown[side]]$
28: $tempDown \leftarrow revNodeList[tempDown[down]]$
29: **end if**
30: **end if**
31: $update(buff, tempDown[lookUpTableIndex])$
32: $PUSH(tempDown)$
33: **end while**
34: **end while**
35: **end procedure**
36: **function** FindCommonPrefixLength(word1,word2)
37: $revWord1 \leftarrow reverse(word1)$
38: $revWord2 \leftarrow reverse(word2)$
39: $length \leftarrow prefixLength(revWord1, revWord2)$
40: $return(length)$
41: **end function**

```
1: function TryCompress(word1,word2,commonPrefixLength)
2: revWord1 ← reversed(word1)
3: revWord2 ← reversed(word2)
4: length ← FindCommonTailLength(revWord1, revWord2, commonPrefixLength)

5: MoveDownAndConnect(revWord1, revWord2, length)
6: end function
7: function FindCommonTailLength(revWord1,revWord2,prefixLength)
8: down1 ← length(resWord1) − prefixLength
9: down2 ← length(resWord2) − prefixLength
10: common ← prefixLength
11: temp1 ← nodeList[0]
12: temp2 ← nodeList[0]
13: down1 ← down1 − 1
14: down2 ← down2 − 1
15: temp1 ← MoveDownTheTree(temp1, down1, revWord1)
16: temp2 ← MoveDownTheTree(temp2, down2, revWord2)
17: matched ← false
18: while matched = false do
19:    temp1 ← MoveDownTheTree(temp1, 1, revWord1)
20:    temp2 ← MoveDownTheTree(temp2, 1, revWord2)
21:    down1 ← down1 + 1
22:    down2 ← down2 + 1
23:    common ← common − 1
24:    matched ← Compare(temp1, temp2)
25: end while
26: return(common)
27: end function
28: function Compare(node1,node2)
29: side1 ← node1[down]
30: side2 ← node2[down]
31: node1 ← nodeList[side1]
32: node2 ← nodeList[side2]
33: while node1[lookUpTableIndex] = node2[lookUpTableIndex] do
34:    node1 ← nodeList[side1]
35:    node2 ← nodeList[side2]
36: end while
37: if ((side1 = null)(side2 = null)) then
38:    return(true)
39: else
40:    return(false)
41: end if
42: end function
```

in the most efficient format.

```
1: function MoveDownTheTree(node,length,word)
2: for i = 1 to length step 1 do
3:    while node[lookUpIndex] ≠ word[i] do
4:       node ← nodeList[node[side]]
5:    end while
6:    node ← nodeList[node[down]]
7: end for
8: end function
9: function MoveDownAndConnect(word1,word2,commonLength)
10: length ← length(word1)
11: for i = 1 to length − commonLength step 1 do
12:    while node1[lookUpIndex] ≠ word1[i] do
13:       node1 ← nodeList[node1[side]]
14:    end while
15:    node1 ← nodeList[node1[down]]
16: end for
17: length ← length(word2)
18: for i = 1 to length − commonLength step 1 do
19:    while node2[lookUpIndex] ≠ word2[i] do
20:       node2 ← nodeList[node2[side]]
21:    end while
22:    node2 ← nodeList[node2[down]]
23: end for
24: node1 ← nodeList[node1[down]]
25: node2[down] ← node1
26: end function
```

Algorithm 4 explains how the compaction is being done. It does this in two steps. In the first step, mark the nodes in a data structure called **forward[]**,indicating which are all nodes have to be moved for the purpose of filling holes. Now every node can look in to this data structure to know whether their pointer fields are going to be changed. If so, they can modify their pointers as mentioned in the forward data structure.In the second phase, we move the nodes safely without loosing the information associated with the nodes. The example in Figure 4.4 explains the procedure applied in the node list representation of trie given above.

Once we have the optimized trie, a memory image of this structure together with look up table could be written to a file from which it could be loaded and used by the front end.

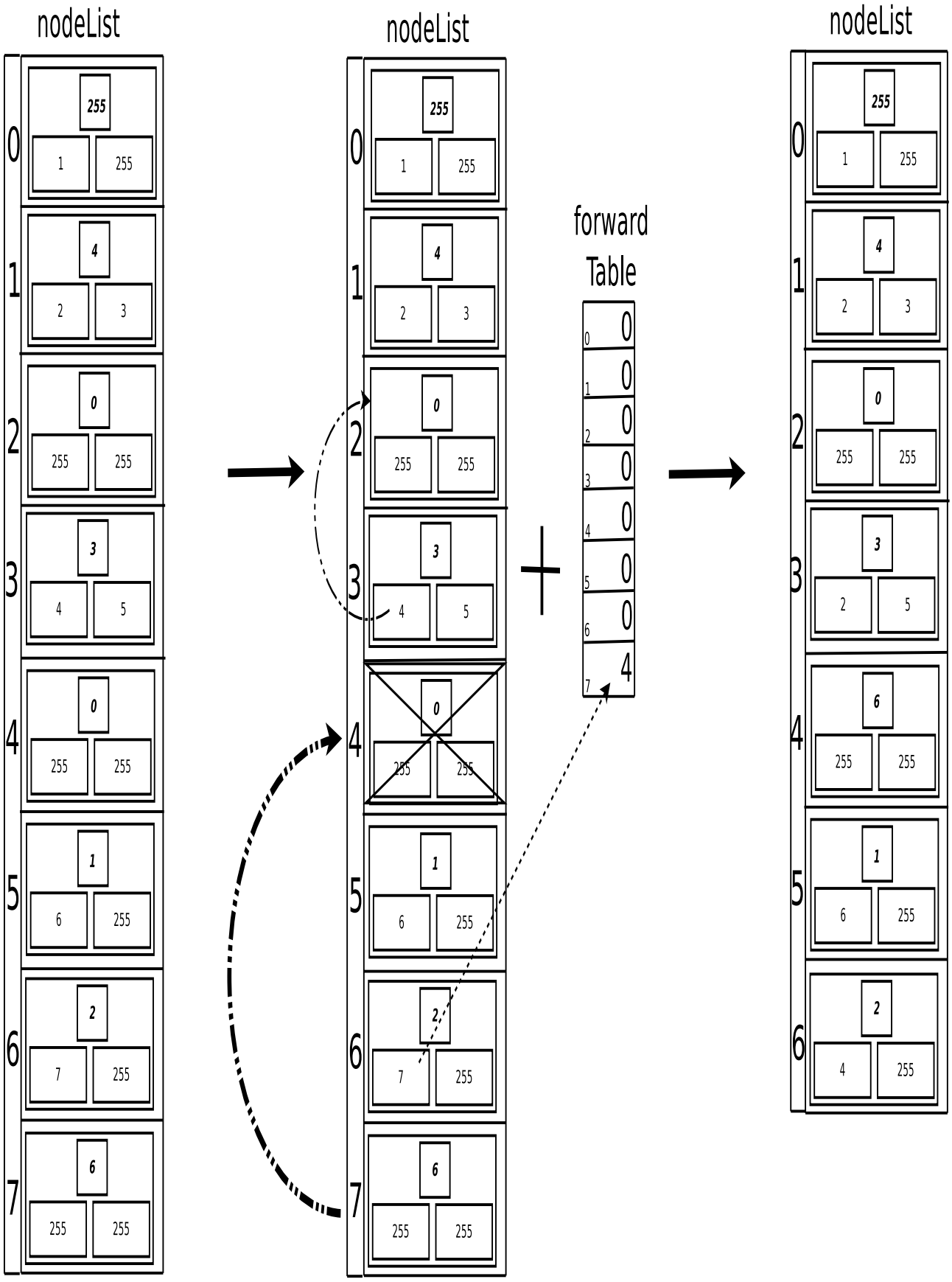

Fig. 4.4: Nodelist represention and sparse removal

Algorithm 4 Algorithm for sparse removal

```
1: procedure RemoveSparse()
Require: nodeList
Ensure: nodeList with all the sparse moved to the end of the list
2: MarkForward()
3: Forward()
4: end procedure
5: function MarkForward()
6: holeIndex ← FindNextHole()
7: for i = nodeCount to 1 step 1 do
8:     if valid(nodeCount[i]) = true then
9:         forward[i] ← holeIndex
10:        holeIndex ← FindNextHole()
11:    end if
12: end for
13: end function
14: function FindNextHole()
15: for i = (currentHoleIndex + 1) to nodeCount step 1 do
16:    if valid(nodeCount[i]) = false then
17:        return(i)
18:    end if
19: end for
20: end function
```

```
1: function Forward()
2: for i = 1 to nodeCount step 1 do
3:    node ← nodeList[i]
4:    if node[side] ≠ null then
5:        node[side] ← forward[node[side]]
6:    end if
7:    if node[down] ≠ null then
8:        node[down] ← forward[node[down]]
9:    end if
10:   if forward[i] ≠ 0 then
11:       exchange(nodeList[forward[i]], nodeList[i])
12:   end if
13: end for
14: end function
```

CHAPTER 5

High Level System Architecture

5.1 System Architecture

This section gives an overall idea about the system(shown in Figure 5.1) by explaining the architectural components and their interrelationships.

Once the user generates a request to open the application, the system starts with performing some preprocessing steps. Then an editor will be shown to the user in which the user can type his message. User provides input to the system using the keypad. The system processes the input, and updates the display so as to show the output back to the user.

During the SMS creation phase, user perform key press, may be with the intention of inputting a new character, or to select one among multiple options provided to him(these may be valid patterns corresponding to previous key press sequence). If the input is a space, that indicates end of current word under processing. In that case, the tool does n't have to do anything. Input alphabets are mapped to buttons 1 to 9. Each button provides a one to many mapping from button to alphabet sub set. For example pressing button 5 means a mapping from key press event of button 5, to a set of Alphabet symbols mapped to button 5. Once the user selects one of these buttons, the system has to identify the alphabets that is mapped to this button, and that suits the situation. To resolve an n^{th} button press, the system uses the results from the previous n-1 button presses and alphabets that are matched to current button, and checks for a combination that matches any of the patterns in the SMS lexicon. If its there, it will be displayed to the user. So formed pattern will be part of the input to the system to resolve the next key press.

In order to parse for a valid pattern, the system keeps track of a two dimensional matrix indicating the alphabet combinations corresponding to a sequence of input key

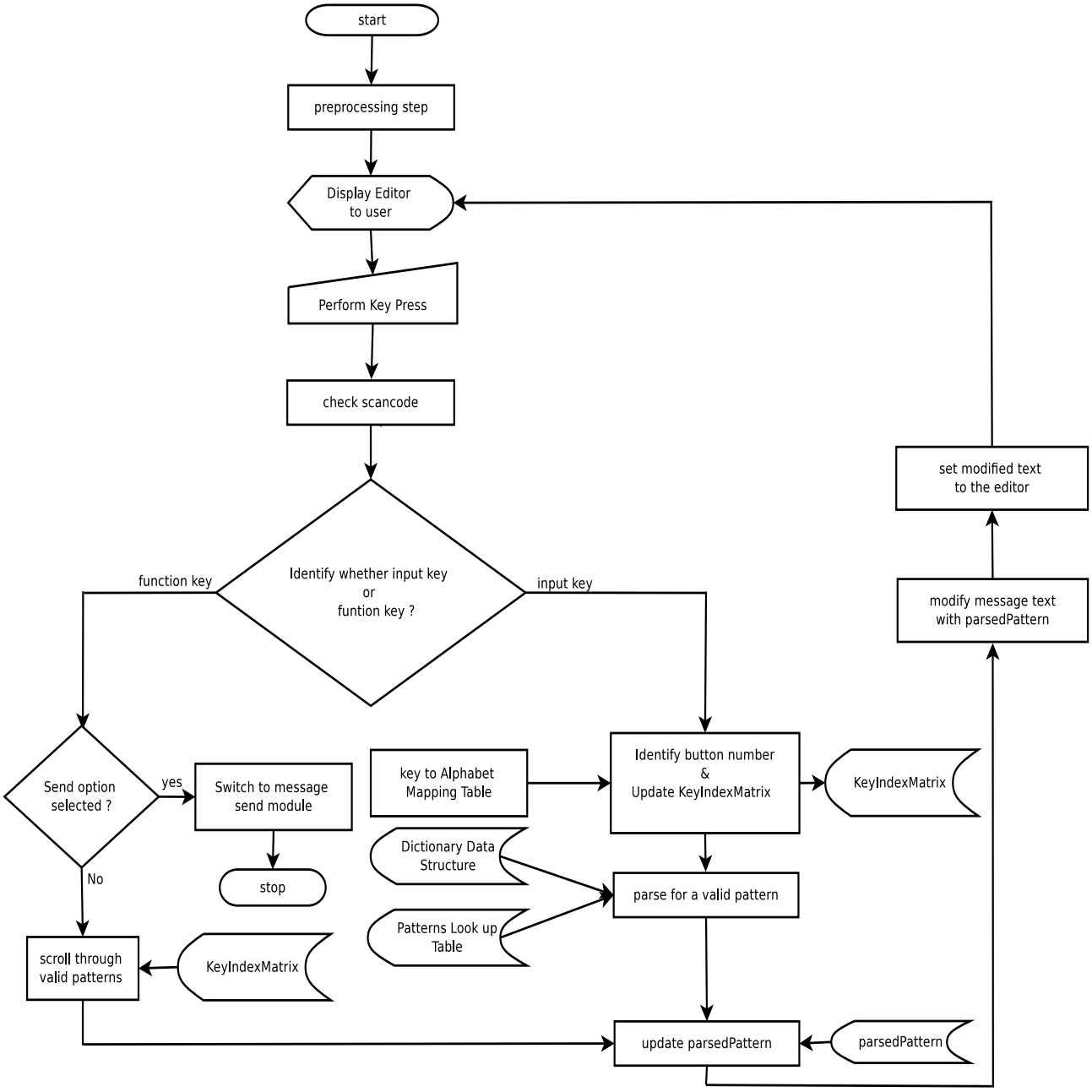

Fig. 5.1: System Architecture

presses. Now the parser module considers the alphabet combinations one by one. Once it finds a valid combination, that will be displayed to the user. Later, it repeats the same process on demand by the user through a function key, which helps the user to rotate through the valid combinations. Parsing the alphabet sequence for the valid check, in the lexicon representation, is a complex procedure shown in Algorithm 5.

Function key initiates a module called *rotate through*. This module plays a supporting role for the tool for ensuring user convenience. At any instance of time, parser may have generated more than one pattern all of which remain valid for that particular combination of key presses. Now the user can *scroll through* those *valid patterns set* so as to select the intended one among them, using the function key.

Once the message is ready, we first transfer the message, and later the control, to native message send module. Now we could use the messaging module services to perform the processes ahead.

5.1.1 Preprocessing Step

The preprocessing step shown in Figure 5.2 is performed when the editor is started.

This phase mainly involves two steps.First, we set up the look up table. Then we should have the lexicon ready in the form of a trie.

As part of the core design process, we develop an efficient representation of both, look up table and the trie (as already discussed) which are stored in a file during preprocessing. Later both data structures are loaded to a section of allotted memory. This will be used by the front end, so that it don't have to regenerate it again. We are following this strategy just because, look up table and trie regeneration from the original lexicon file, are very much time consuming tasks,which cannot be done during preprocessing. Now whenever needed, the front end just loads it and uses it, not being aware about where it comes from.

Algorithm 5 Algorithm for parsing the valid pattern in Lexicon

1: **procedure** parseForPattern()
Require: keyIndexlist:sequence of key presses to be analyzed
Ensure: parsePattern:One of the successfully parsed patterns.

2: $state \leftarrow InitParse(1)$
3: **if** $state = true$ **then**
4: $display(parsePattern)$
5: **end if**
6: **end function**
7: **function** InitParse(level)
8: **for** $i = 1$**to** $infinity$**step** 1 **do**
9: **if** $checkInLexicon(level)$ **then**
10: **if** $level = patternLength$ **then**
11: $return(true)$
12: **else**
13: $InitParse(level + 1)$
14: $return(true)$
15: **end if**
16: **end if**
17: $next(buttonIndex[level - 1])$
18: **end for**
19: **end function**
20: **function** CheckInLexicon(level)
21: **for** $i = 1$**to** $patternLength$**step** 1 **do**
22: $parsePattern[i] \leftarrow Key[KeyIndexList[i]][buttonIndex[i]]$
23: $ModifyMathrasParsePattern[i]$
24: **end for**
25: $parsedFlag \leftarrow 1$
26: $node \leftarrow nodeList[0]$
27: $downTemp \leftarrow node[down]$
28: **while** $(downTemp \neq null) and (parsedFlag = 1)$ **do**
29: $index \leftarrow FindPrefix(parsePattern, i, patternLength)$
30: $downPtr \leftarrow downTemp$
31: **while** $downPtr[lookupTableIndex] \neq index$ **do**
32: $downPtr \leftarrow downPtr[side]$
33: **end while**
34: **if** $down[lookUpTableindex] = index$ **then**
35: $parsedFlag \leftarrow 1$
36: $downPtr \leftarrow downPtr[down]$
37: **end if**
38: **end while**
39: **end function**

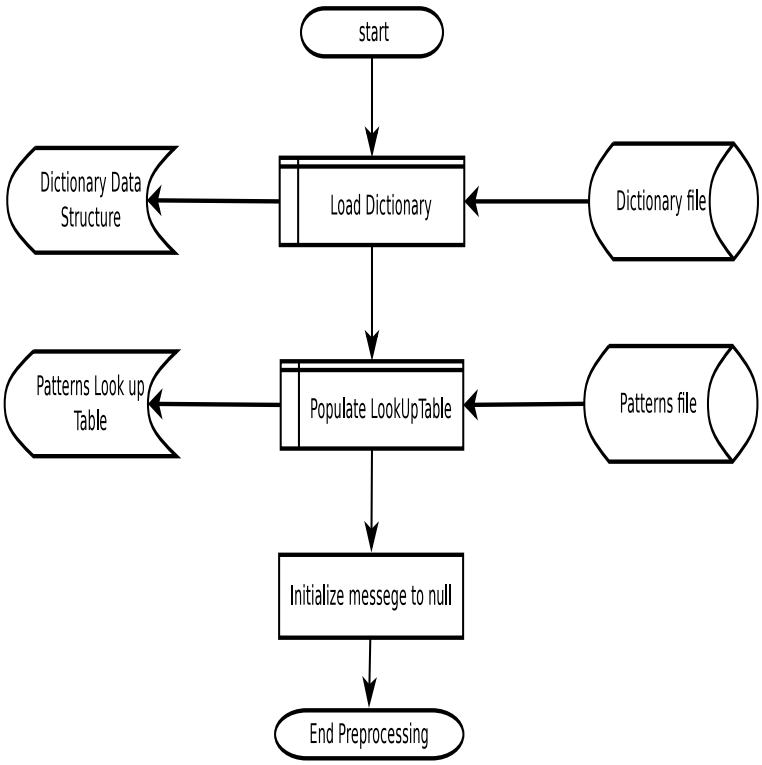

Fig. 5.2: Preprocessing Phase

5.1.2 Parsing Module

This module (shown in Algorithm 5) uses LookUpTable data structure and lexicon data structure(in the form of nodeList) for the purpose of performing validity check for a possible pattern. Other than that the following book keeping information will be used by the algorithm.

- KeyN[1...C] - Represents C alphabets corresponding to Key N.
- Mathra[1...count][0...1] - Mathra[currentCount][0] represents the symbol used when they occur as first alphabet in a word and Mathra[currentCount][1] represents the symbol used, when they occur in between.
- keyIndexList[i] - Button Index corresponding to i th key press.
- buttonIndex[i] - Index of alphabet under consideration, for button ith pressed button.

- patternLength - Length of input key sequence.

```
1: procedure parseForPattern()
2: function FindPrefix(word,index,length)
3: for i = index to length step 1 do
4:    for i = j to patternCount step 1 do
5:       pattern ← LookupTable[j]
6:       if word[i...(i + length)] = pattern then
7:          return(j)
8:       end if
9:    end for
10: end for
11: end function
```

5.1.3 The Scroll through Module

As already mentioned, the rotate through module shown in Algorithm 6 does the job of facilitating user's task of selecting one among several valid patterns.It also uses the book keeping information as in ParseForPattern() algorithm.

Algorithm 6 Algorithm for scrolling through the list of valid patterns

```
1: procedure Scroll
2: function ScrollThrough(level)
3: for i = 1 to infinity step 1 do
4:    if checkInLexicon(level) then
5:       if level = patternLength then
6:          return(true)
7:       else
8:          ScrollThrough(level + 1)
9:          return(true)
10:      end if
11:   end if
12:   next(buttonIndex[level − 1])
13: end for
14: end function
```

5.2 Keypad Design

We were provided with an efficient button to keypad mapping for Hindi. The design of such keypads would have been focused on reducing conflicts across alphabets mapped

to same key.

Fig. 5.3: Keypad design

5.3 Keypad Event Handling

We override the Key Event handler and capture the keys in order to make the system respond in the way we want. During a press,*input method key event* has been generated, which we simply override. We capture the *release key event*, process it and pass the modified value to our system according to the keypad design mentioned above.

This design differs from the keypad design available in traditional mobiles. In traditional mobiles, with the dictionary feature disabled, if we want to press i^{th} alphabet on the j^{th} button, we have to press the button j, i times within a short time period. Because of this timing difficulty, our design demands user to generate a button click sequence j i if he want to type i^{th} character on the jth button. If the dictionary feature enabled, our keypad works like a normal keypad.

CHAPTER 6

Implementation and Results

6.1 Overview

In this section we cover the translation of designed aspects in to implementation.On the way to final product we had to face a lot of dilemmas.

6.2 Input

A set of 7739 Hindi words and an optimized key pad design.

6.3 Fixing Back end Parameters

The look up table size fixed to be 256. Maximum pattern size to be 5. We fix the benefit factor in pattern generation to be 44. We started with a value 50 and tuned it till the value reached 44 where we got the best fit of look up table with the input patterns.

6.4 Environment

The target platform was defined as part of requirement. We were asked to develop the tool for **S60 model** mobile phones, which has **Symbian** as the mobile operating system. The task application development was well supported by tools provided by Nokia. This set includes,

- Carbide .C++ IDE version 2.3 -This is the IDE on which we develop programs in Symbian C++.

- Nokia N97 emulator version 1.0 - emulator for testing the proper working of code without using the actual device.
- Active Perl version 5.10.1 - Supporting tool for program development. Bridges the gap between IDE and Emulator.
- Qt for Symbian version 4.6.2- Extension to the current IDE with a very useful collection of APIs.
- target device - Nokia n97 mini.

The whole kit of packages got installed on the Windows platform, where we performed the front end development.

We implemented the core feature, which is storing the lexicon in a well contained data structure which meets our needs. The data structure and look up table are written to a file in the form primary memory image. This works because we do not use pointers. Now the code developed in Symbian C++, with Qt support [8] on Windows platform, generates executable which uses this file to meet user requirement of performing word prediction. We had to convert the file in big endian format to little endian format because of the platform difference. Further compression was done on the UTF-8 codes of the alphabet for better memory optimization.

6.5 Front end

We developed the editor using the QTextEdit class object in Qt. QFile class deals with file operations. It has functions like, open(), close(),read() etc. QTextedit class is associated with an Event handler which we had to override to implement the editing task. We read the scan code associated with the key event to identify it uniquely. Events other than key events will be bypassed. The QString class made the job dealing with strings easier. In the installed emulator, we put the lexicon file and look up table file into a folder so as to simulate the file to be residing in mobile.The files are in the form of a collection of hex codes corresponding to the alphabets for Hindi language.

There were 53 alphabet symbols mapped to 9 keys from 1 to 9. Zero meant for space character.

6.6 Results

The following table 6.1 discusses space consumption characteristics of the data structure.

Field	Value
Number of entries in LUT	256
Maximum pattern length	5
LUT size	845 bytes
Size of a node in the data structure	5 bytes
Number of nodes	8374
Total data memory	42KB

Table 6.1: Space consumption characteristics of the data structure

CHAPTER 7

Conclusion, Limitations and Future Work

7.1 Overview

This chapter concludes the presentation and covers the limitations associated with the system.

7.2 Dealing with the Problem

We started approaching the problem by looking into similar work, where on most of the occasions, tries were the data structures, taking up the jobs which are so close to what we want. So we decided to start with this standard data structure. As the work progressed, the data structure kept on evolving. New features got added to the base structure.

The data structures we develop consist of,

- Look Up Table
- A specialized version of trie

Once the data structures got deployed, then it was about porting it so as to make the job done completely.

The front end tasks were done on a windows platform, where the emulator was installed. The Qt tool made the task of handling GUI easier. It even provided event handling APIs which we easily modified to meet our purpose. In the end, it was not a tough job to have a nice application running on a nice platform.

7.3 Benefits

- A tool which supports sending of Hindi text SMS.
- Small memory requirement makes it possible to be installed on low end mobiles.
- Design is possible to be reused for development of SMS Lexicon in any language.

7.4 Limitations

The tool does not have maintainability. We can't add new words to it. There are dictionary data structures that let you to do both addition and removal of words [9]. Both are possible by extending this design.Input lexicon could have been optimized based on user's frequency of usage of words. Another limitation is numbers and additional symbols can not be part of the input message.

7.5 Future Work

The above mentioned short comings should be overcome in the future. Instead of pattern prediction, we could target word prediction, and that is even could be based on its position in the target sentence. We could extend the application so to make it a target language independent word predictor. It just has to be equipped with code tables and key pad mappings for all languages. So that later, it could be loaded and unloaded with word sets for various supported target language, depending on the need.

REFERENCES

[1] http://en.wikipedia.org/wiki/T9_(predictive_text).

[2] R. L. R. Thomas H Cormen, Charles E Leiserson, *Introduction to Algorithms*. 2001. http://www.introtoalgorithms.com.

[3] http://www.pcworld.com/businesscenter/article/195025/india_adds_record%_20_million_mobile_subscribers_in_march.html.

[4] http://www.pluggd.in/india-mobile-market/report-sms-as-a-vas-service-297.

[5] G. Janssen, "Design of a pointerles bdd package," International Workshop on Logic Synthesis (IWLS), 2001.

[6] R. Bloor, "Essential s60 developers' guide." Symbian software limited. http://www.forum.nokia.com/info/sw.nokia.com/id/80dc01fa-2260-49ca-8ee3-f0a414adb78a/Essential_S60_Developers_Guide.pdf.html.

[7] R. Harrison, "Symbian os c++ for mobile phones." Symbian software limited, 2003. http://www.forum.nokia.com/info/sw.nokia.com/id/80dc01fa-2260-49ca-8ee3-f0a414adb78a/Essential_S60_Developers_Guide.pdf.html.

[8] http://symbianresources.com/tutorials/qt.php.

[9] S. Ristov, "Lz trie and dictionary compression," 2002. http://www.introtoalgorithms.com.