FINITE STATE RECOGNIZER AND STRING SIMILARITY BASED SPELLING CHECKER FOR BANGLA

Submitted to A Thesis

The Department of Computer Science and Engineering

of

BRAC University

by

Munshi Asadullah

In Partial Fulfillment of the Requirements for the Degree

of

Bachelor of Science in Computer Science and Engineering

Fall 2007

BRAC University, Dhaka, Bangladesh

D E C L A R A T I O N

I hereby declare that this thesis is based on the results found by me. Materials of work found by other researcher are mentioned by reference. This thesis, neither in whole nor in part, has been previously submitted for any degree.

---------- ----------

Signature of Supervisor Signature of Author

A C K N O W L E D G E M E N T S

Special thanks to my supervisor Mumit Khan without whom this work would have been very difficult. Thanks to Zahurul Islam for providing all the support that was required for this work. Also special thanks to the members of CRBLP at BRAC University, who has managed to take out some time from their busy schedule to support, help and give feedback on the implementation of this work.

A b s t r a c t

A crucial figure of merit for a spelling checker is not just whether it can detect misspelled words, but also in how it ranks the suggestions for the word. Spelling checker algorithms using edit distance methods tend to produce a large number of possibilities for misspelled words. We propose an alternative approach to checking the spelling of Bangla text that uses a finite state automaton (FSA) to probabilistically create the suggestion list for a misspelled word. FSA has proven to be an effective method for problems requiring probabilistic solution and high error tolerance. We start by using a finite state representation for all the words in the Bangla dictionary; the algorithm then uses the state tables to test a string, and in case of an erroneous string, try to find all possible solutions by attempting singular and multi-step transitions to consume one or more characters and using the subsequent characters as look-ahead; and finally, we use backtracking to add each possible solution to the suggestion list. The use of finite state representation for the word implies that the algorithm is much more efficient in the case of non-inflected forms; in case of nouns, it is even more significant as Bangla nouns are heavily used in the non-inflected form. In terms of error detection and correction, the algorithm uses the statistics of Bangla error pattern and thus produces a small number of significant suggestions. One notable limitation is the inability to handle transposition errors as a single edit distance errors. This is not as significant as it may seem since the number of transposition errors are not as common as other errors in Bangla. This paper presents the structure and the algorithm to implement a Practical Bangla spell-checker, and discusses the results obtained from the prototype implementation.

Table of Contents

List of Figures

List of Tables

Introduction

Bangla is a complex language with a large lexicon, and it is primarily a close-phonetic oriented language. A morphologically-based Bangla spelling checker using string pattern comparison often produces huge number of possible solutions due to the use of compound characters in Bangla. So, phonetic based spell checking is the better approach for Bangla. Puspa (UzZaman and Khan, 2006) is one such phonetic spelling checker for Bangla, which, like any phonetic based spell checkers, has its limitations. It often fails to suggest for multiple errors in both the root and the suffix of a word. It also produces huge number of suggestion for detectable multiple error situations. But, as it is necessary to have minimal dictionary size for a spell-checker, it is also important that it produces small number of relevant suggestions for a word with around 50% error.

Error tolerant finite state recognition uses its underlying finite-state recognizer to relate an erroneous word with is most probable solution set. Our goal is to develop a system that can handle non-word errors in Bangla efficiently and quickly. In our proposed system, all the possibilities are addressed while checking a non-word error. The reason is that every character may be correct and at the same time may be wrong, and so the better approach is to consider all possibilities and put a threshold value to control the result size. The threshold refers to the minimum edit-distance. The primary state table generation ensures that only the words available in the dictionary can be produced and nothing else. The motivation behind the study is to come up with a strong spellchecking algorithm using the finite state machine to produce better result from the existing spell-checkers.

In this paper, we tried to show the improvements achieved by using Finite State approach and the suggested algorithm and listing the results yielded by our working prototype. Our strategy is to map all the roots and their inflected forms in a state table and use it with the algorithm to check different types of error firstly as a single character error, as an earlier study (Kundu and Chaudhuri, 1999 – Chaudhuri, 2001) suggested that error length of 1 to 2 is about 74% of all Bangla non-word errors, using the current character and a look-ahead character. If the error turns out to be a multi-character error, multiple transitions for both the current character and the look-ahead character will be made. If a solution can not be found then the current states will be saved and all the previous characters are inserted, and a multi-transition to consume the current character from the start state will be attempted. The final attempt may lead to a distant solution but as it will have a large edit-distance; it may eventually be left out of the solution list. On the other hand, saving the current state before the desperate approach actually gives the ability to even check 3 and more character error with the subsequent characters. This should cover the vast majority of the spelling errors as 1 to 3 character errors make up 90.8% of all Bangla non-word errors (Kundu and Chaudhuri, 1999 – Chaudhuri, 2001).

Related Works

There has been some activity in designing spelling checkers for Bangla and other South Asian languages. One recent example is the work done by UzZaman and Khan on Bangla spelling checker that uses double-metaphone called Puspa speller (UzZaman and Khan, 2006). Approximate strings matching algorithms (Choudhury, 2001) and a direct dictionary look up method (Abdullah and Rahman, 2004) have been used so far for the detection of typographical errors and cognitive phonetic errors. Other many work have done on this field or in the related field. One significant work has done on Error-tolerant Finite State Recognition with Applications to Morphological Analysis and Spelling (Oflazer, 1996). Another significant work on Exploring the limits of Spellcheckers: A Comparative Study in Bengali and English (Bhatt, Choudhury, Sarkar and Basu, 2005) gives a realistic method for evaluating spelling checkers.

Section 1: General Overview

Section 1: General Overview

1.1. Natural Language Processing (NLP)

1.1.1. Definition

'Natural Language' (NL) is any of the languages naturally used by humans, i.e. not an artificial or man-made language such as a programming language. 'Natural language processing' (NLP) is a convenient description for all attempts to use computers to process natural language.

Natural language processing (NLP) is a subfield of artificial intelligence and linguistics. It studies the problems of automated generation and understanding of natural human languages. Natural language generation systems convert information from computer databases into normal-sounding human language, and natural language understanding systems convert samples of human language into more formal representations that are easier for computer programs to manipulate.

1.1.2. Scope of NLP

Over the last few decades NLP manage its distinct identity over the vast computational spectrum, today's computers are working with. The major fields of interest and attention can be categorize are as follows,

- **Speech synthesis:** although this may not at first sight appear very 'intelligent', the synthesis of natural-sounding speech is technically complex and almost certainly requires some 'understanding' of what is being spoken to ensure, for example, correct intonation.
- **Speech recognition:** Basically the reduction of continuous sound waves to discrete words. It is currently a dominant field in the field of NLP.
- **Natural language understanding:** Moving from isolated words (either written or determined via speech recognition) to 'meaning'. This may involve complete model systems or 'front-ends', driving other programs by NL commands.
- **Natural language generation:** Generating appropriate NL responses to unpredictable inputs. A question answering system is a perfect example of such a system
- **Text categorization:** Analysis of documents and categorize them according to its context may seem to be a great idea when thousands of people are surfing the web for thousands of different things but it actually is more helpful in the field if machine learning that works with other contexts of NLP research.
- **Imperfect or irregular input:** Foreign or regional accents and vocal impediments in speech; typing or grammatical errors, OCR errors in texts are also a necessary area to work with in NLP.
- **Machine translation (MT):** translating one NL into another is one of the largest areas of interest for NLP. A Lot of work is currently going on in this field.

1.2. Word Spelling Error

1.2.1. Word Spelling Error

The word-error can belong to one of the two distinct categories, namely, non-word error and real-word error. Let a string of characters separated by spaces or

punctuation marks be called a candidate string. A candidate string is a valid word if it has a meaning. Else, it is a non-word. By real-word error we mean a valid but not the intended word in the sentence, thus making the sentence syntactically or semantically ill-formed or incorrect. In both cases the problem is to detect the erroneous word and either suggest correct alternatives or automatically replace it by the appropriate word.

1.2.2. Word Error Pattern

Errors occurred in a text depends on several factors. Of those the most important ones are Human Factor that is, if the writer do not know the spelling or mistyped something, Source of the document that is, whether it is a handwritten Text or Typed on some system etc. If handwritten then which OCR system converts it to digital text is also very important. Most OCR sources make transcription error and humans written sources also make such mistakes as well. Although the error type totally depends on the source of the actual document. The errors made in handwritten text are often different or at least occurs in a different rate than in a typed text. The errors that are in the actual document are similar to the errors made by human.

The errors that are made purely by an OCR system are mostly pattern recognition error. It is very common to have a substitution of a character with a similar looking character (e.g. ‘e’ and ‘c’ or ‘m’ and ‘nn’). Considering the source of the actual document any other types of errors may also occur. OCR systems also produce some unusual patterns that lead to totally improbable errors.

For typed text language and keyboard layout holds the key to possible error type. The generalized error type classification for the text written by a human is as follows,

- **Cognitive Error:** When the actual spelling is unknown to the writer and the writer tries to write it with probable spelling, this type of error occurs.
- **Phonetic Error:** This is a special type of cognitive error. Usually humans try to substitute an unknown spelling with the most probable spelling that sounds somewhat similar to the actual word. Often similar sounding

characters are substituted in this type of error (e.g. ‘f’ for ‘ph’ or ‘c’ for ‘k’). e.g. “card” replaced by “kurd”.

- **Typographical Error:** Typos due to slip of neuron motor coordination or may be just because a finger was placed in the wrong position usually is the reason behind this type of error. This type of error can primarily be seen in Typed text. The error patterns here are somewhat dependent on the language and keyboard layout. The errors usually be seen can be categorize as follow,
  - **Substitution Error:** When the desired character in a word got replaced by some other character that is the actual character is substituted. e.g. Pain → Psin.
  - **Insertion Error:** While writing a word if a character is pressed twice or some other character is pressed along with the original character, the undesired character is considered to be inserted. e.g. Pain → Paiin.
  - **Deletion Error:** While writing a word if any or more character is missing then it is called to be deleted and the error is considered to be a deletion error. e.g. Painful → Painfl.
  - **Transposition Error:** As the name suggest, transposition errors occur when characters have "transposed" - that is, they have switched places. Transposition errors are almost always human in origin. The most common way for characters to be transposed is when a user is touch typing at a speed that makes them input one character, before the other. This may be caused by their brain being one step ahead of their body. e.g. Present → Presetn.

All these errors can produce both real- word and non-word errors. Real-word errors are more difficult to detect than non-word errors because the former requires deep or shallow cortex analysis.

1.2.3. Word Error Model

In the statistical spell checking language model gives the probabilistic nature of a correction and also the probability of that suggestion to be produced from the given wrong word. This paper though do not suggest a statistical approach towards spell checking, It dose get a lot of help from the error pattern study. The

model this paper adopts is the analysis of error pattern and thus optimizing the underlying algorithm accordingly.

The features that are used in this model are the rate of each type of error to occur and the rate of how many errors could occur in a word. The underlying assumption is the error types and possible error count handling in the descending sequence of their rate of occurrence should make the system work faster.

1.2.4. Error Detection

Isolated word-error detection techniques can use a wide range of techniques. The most common though is the dictionary lookup. With a wordlist with sufficient coverage of a language and a fast dictionary lookup (typically hashing) often provides the simplest solution to the non-word error detection problem.

However this basic technique does not scale up well for a morphologically rich language, where a deep morphological analysis is might be necessary to avoid false errors, i.e. in this method proper nouns are often misclassified as non-words.

N-gram based spellcheckers does overcome some of these problems but occasionally might fail to detect non-word errors. Real word error detection is rather complicated and requires at least some shallow morphological knowledge is necessary. Although, the use of word n-gram or POS-N-Gram is used in the statistical approaches, complete understanding of the text is necessary to achieve a high accuracy.

1.2.5. Error Correction

There are different isolated-word error correction techniques based on minimum edit distance, similarity keys, rule based on error patterns, n-grams, probabilistic and soft computing techniques. Among those this paper is on a method that uses edit distance as its correction mechanism and thus edit-distance approach is described in details in the later part of the paper.

The motivation behind the similarity key and some of the rule based system is to capture and correct phonetic errors. Most typographic errors can be detected and corrected by the edit-distance methods. Sometimes the keyboard layout is also considered to get improved output.

Just as in the case of error detection, error correction in real-word error is also required detailed syntactic or symmetric analysis of the text and as it is off the context of this paper, it will not be discussed.

1.3. Spelling Checker

1.3.1. Spelling Checker

In computing terms, a spell checker or spelling checker is a design feature or a software program designed to verify the spelling of words in a document, query, or other context, helping a user to ensure correct spelling. A spell checker may be implemented as a stand-alone application capable of operating on a block of text. Spelling checkers are most often implemented as a feature of a larger application, such as a word processor, email client, electronic dictionary or search engine.

Simple spelling checkers operate at the word level, by comparing each word in a given input against a vocabulary (often erroneously referred to as a dictionary). If the word is not found within the vocabulary, it is designated erroneous, and algorithms may be run to detect which word the user most likely meant to type. One simple such algorithm is listing words from the dictionary with a small Levenshtein distance from the typed word.

Spelling checkers can operate as the user enters text, notifying the user when an error is made (usually by underlining the erroneous text). They can also operate at the user's request, checking an entire document or email at once. A word processor will typically offer both modes of operation.

Many spelling checkers can operate in more than one language. There are many cases in which a user may intentionally type a word which is not within the vocabulary of the language in which the spelling checker is operating; proper

nouns and acronyms are two common examples. To solve this problem, most spelling checkers allow the user to add custom words to the spelling checker's vocabulary. Usually the user also has the option to ignore specific errors.

1.3.2. History

The first spelling checkers were widely available on mainframe computers in the late 1970s. The first spelling checkers for personal computers appeared for CP/M computers in 1980, followed by packages for the IBM PC after it was introduced in 1981. Developers such as Maria Mariani, Soft-Art, Microlytics, Proximity, Circle Noetics, and Reference Software rushed OEM packages or end-user products into the rapidly expanding software market, primarily for the PC but also for Apple Macintosh, VAX, and UNIX. On the PCs, these spelling checkers were standalone programs, many of which could be run in TSR mode from within word-processing packages on PCs with sufficient memory.

However, the market for standalone packages was short-lived, as by the mid 1980s developers of popular word-processing packages like WordStar and WordPerfect had incorporated spelling checkers in their packages, mostly licensed from the above companies, who quickly expanded support from just English to European and eventually even Asian languages. However, this required increasing sophistication in the morphology routines of the software, particularly with regard to heavily-inflected languages like Hungarian and Finnish. Although the size of the word-processing market in a country like Iceland might not have justified the investment of implementing a spelling checker, companies like WordPerfect nonetheless strove to localize their software for as many as possible national markets as part of their global marketing strategy.

Recently, spell checking has moved beyond word processors as Firefox 2.0, a web browser, has spell check support for user-written content, such as when writing on many webmail sites, blogs, and social networking websites. The web browser Opera and the instant messaging client Gaim also offer spell checking support, transparently using GNU Aspell as their engine.

1.3.3. Design Paradigm

As already outlined, a spelling checker customarily consists of two parts: A set of routines for scanning text and extracting words, and A wordlist (the vocabulary; often referred to as a dictionary) against which the words found in the text are compared.
The scanning routines sometimes include language-dependent algorithms for handling morphology. Even for a lightly inflected language like English, word extraction routines will need to handle such phenomena as contractions and possessives. It is unclear whether morphological analysis provides a significant benefit.

The wordlist might simply be a list of words, or it might also contain additional information, such as hyphenation points or lexical and grammatical attributes.
As an adjunct to these two components, the program's user interface will allow users to approve replacements and modify the program's operation.

One exception to the above paradigm is spelling checkers which use solely statistics, such as n-grams, but these have never caught on. In some cases spell checkers use a fixed list of misspellings and suggestions for those misspellings; this less flexible approach is often used in paper-based correction methods, such as the see also entries of encyclopedias.

1.3.4. Phonological Approach

There are several ways of looking up a word from a dictionary. One can use a flat file to keep the dictionary or use some indexed data structure or even some DBMS. A different approach though is to use Phonological Coding of a word to check its correctness and generating suggestion. The two major encodings are, **Soundex** and **Metaphone**. These encoding map the words to there compact phonetic signature and very useful to detect phonetic errors. The detail is necessary because the method discussed in this paper can be incorporated with the phonological approach.

The first step is to encode all the words the dictionary has into its corresponding phonological code. The two algorithms that are most commonly used are,

- **Soundex:** Soundex is a phonetic algorithm for indexing names by their sound when pronounced in English. The basic aim is for names with the same pronunciation to be encoded to the same string so that matching can occur despite minor differences in spelling. Soundex is the most widely known of all phonetic algorithms and is often used (incorrectly) as a synonym for "phonetic algorithm". The Soundex code for a name consists of a letter followed by three numbers: the letter is the first letter of the name, and the numbers encode the remaining consonants. Similar sounding consonants share the same number so, for example, the labial B, F, P and V are all encoded as 1. Vowels can affect the coding, but are never coded directly unless they appear at the start of the name. The exact algorithm is as follows:
  - Retain the first letter of the string.
  - Remove all occurrences of the following letters, unless it is the first letter: a, e, h, i, o, u, w, y
  - Assign numbers to the remaining letters (after the first) as follows: b, f, p, v = 1 | c, g, j, k, q, s, x, z = 2 | d, t = 3 | l = 4 | m, n = 5 | r = 6.
  - If two or more letters with the same number were adjacent in the original name (before step 1), or adjacent except for any intervening h and w (American census only), then omit all but the first.
  - Return the first four characters, right-padding with zeroes if there are fewer than four.

- **Metaphone:** Metaphone is a phonetic algorithm, an algorithm for indexing words by their sound, when pronounced in English. Metaphone was developed by Lawrence Philips as a response to deficiencies in the Soundex algorithm. It is more accurate than Soundex because it "understands" the basic rules of English pronunciation. The original author later produced a new version of the algorithm, which he named Double Metaphone that produces more accurate results than the original algorithm. The algorithm produces keys as its output. Similar sounding words share the same keys and are of variable length. A double key bases more accurate algorithm called Double Metaphone is also available and being used.

1.3.5. Some Major GNU Spelling Checkers

Our prototype is designed as a open source application and a practical product can be developed under GPL, thus some available systems needs to be discussed.

- **Ispell:** Ispell is a spelling checker for UNIX that supports most Western languages. It offers several interfaces, including a programmatic interface for use by editors such as emacs. Unlike GNU Aspell, ispell will only suggest corrections that are based on a Damerau-Levenshtein distance of 1; it will not attempt to guess more distant corrections based on English pronunciation rules.

  Ispell has a very long history that can be traced back to a program that was originally written in 1971 in PDP-10 Assembly language by R.E.Gorin, and later ported to the C programming language and expanded by many others. The generalized affix description system introduced by ispell has since been imitated by other spelling checkers such as MySpell.

  Like most computerized spelling checkers, ispell works by reading an input file word by word, stopping when a word is not found in its dictionary. Ispell then attempts to generate a list of possible corrections and presents the incorrect word and any suggests to the user, who can then choose a correction, replace the word with a new one, leave it unchanged, or add it to the dictionary.

  Ispell pioneered the idea of a programmatic interface, which was originally intended for use by emacs. Other applications have since used the feature to add spelling checking to their own interface, and GNU Aspell has adopted the same interface so that it can be used with the same set of applications. There are ispell dictionaries for most widely-spoken Western languages.
- **GNU Aspell:** GNU Aspell, usually called just Aspell, is a free software spell checker designed to replace Ispell. It is the standard spelling checker software for the GNU software system. It also compiles for other Unix-like operating systems and Microsoft Windows. The main program is licensed under the GNU Lesser General Public License (GNU LGPL), the documentation

under GNU Free Documentation License (GNU FDL). Dictionaries for it are available for about 70 languages. The main maintainer is Kevin Atkinson.

- **Hunspell:** Hunspell is a spell checker and morphological analyzer designed for languages with rich morphology and complex word compounding or character encoding, originally designed for Hungarian language. It has been the default spell checker of OpenOffice.org office suite since March 2006, and is being incorporated into Mozilla products, such as Mozilla Thunderbird and Mozilla Firefox. Hunspell's code base comes from MySpell spell checker. Hunspell is free software, distributed under the terms of a GPL/LGPL/MPL tri-license.
- **MySpell:** MySpell was started by Kevin Hendricks to integrate various open source spelling checkers into the OpenOffice.org build. With a little prodding from Kevin Atkinson, the author of Pspell and Aspell, a new spelling checker (MySpell) was written in C++ that supported affix compression, based on Ispell. This code has been given to Pspell and will eventually make it into a future Pspell release after it has been integrated.

1.4. Finite State Machine

1.4.1. Finite State Automation

Finite State Machines (FSM), also known as Finite State Automation (FSA), at their simplest, are models of the behaviors of a system or a complex object, with a limited number of defined conditions or modes, where mode transitions change with circumstance. Finite state machines consist of 4 main elements:

- States which define behavior and they may produce actions.
- State transitions, which are movement from one state to another.
- Rules or conditions that must be met to allow a state transition.
- Input events which are either externally or internally generated, which may possibly trigger rules and lead to state transitions.

A finite state machine must have an initial state which provides a starting point, and a current state which remembers the product of the last state transition. Received input events act as triggers, which cause an evaluation of some kind of the rules that govern the transitions from the current state to other states. The best way to visualize a FSM is to think of it as a flow chart or a directed graph of

states, though as will be shown; there are more accurate abstract modeling techniques that can be used.

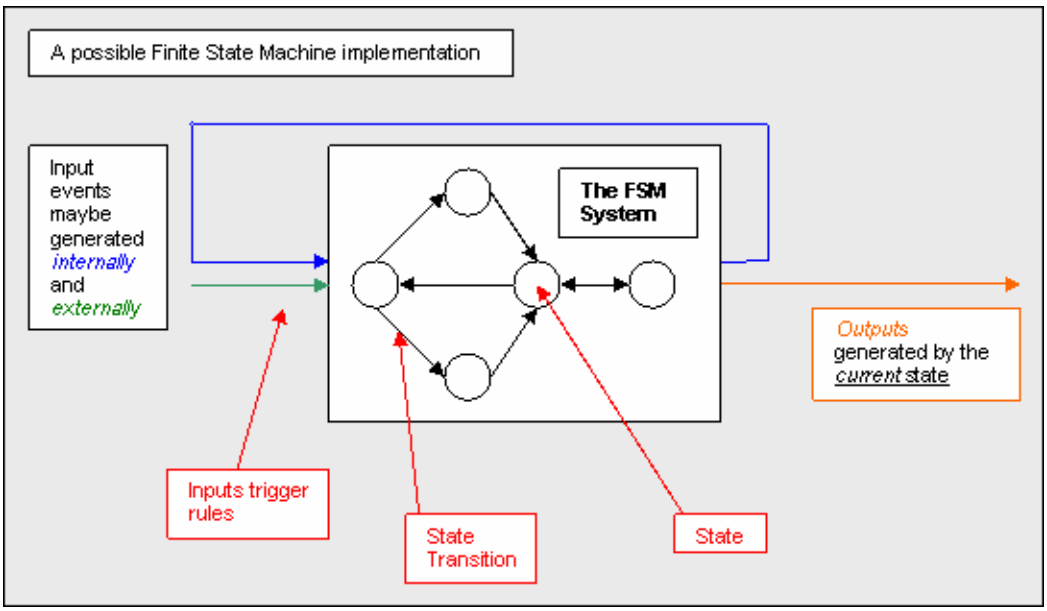


figure 1.1: A possible finite state machine implementation

FSM is typically used as a type of control system where knowledge is represented in the states, and actions are constrained by rules.

"...One of the most fascinating things about FSMs is that the very same design techniques can be used for designing Visual Basic programs, logic circuits or firmware for a microcontroller. Many computers and microprocessor chips have, at their hearts, a FSM."

Finite state machines are an adopted artificial intelligence technique which originated in the field of mathematics, initially used for language representation. It is closely related to other fundamental knowledge representation techniques which are worth mentioning, such as semantic networks and an extension of semantic networks called state space.

Finite state machines is not a new technique, it has been around for a long time. The concept of decomposition should be familiar to people with programming or design experience. There are a number of abstract modeling techniques that may help or spark understanding in the definition and design of a finite state machine, most come from the area of design or mathematics.

Like most techniques, heuristics for when and how to implement finite state machines are subjective and problem specific. It is clear that FSMs are well suited to problems domains that are easily expressed using a flow chart and possess a set of well defined states and rules to govern state transitions.

1.4.2. Advantages of FSM

- Their simplicity make it easy for inexperienced developers to implement with little to no extra knowledge (low entry level)
- Predictability (in deterministic FSM), given a set of inputs and a known current state, the state transition can be predicted, allowing for easy testing
- Due to their simplicity, FSMs are quick to design, quick to implement and quick in execution
- FSM is an old knowledge representation and system modeling technique, and its been around for a long time, as such it is well proven even as an artificial intelligence technique, with lots of examples to learn from
- FSMs are relatively flexible. There are a number of ways to implement a FSM based system in terms of topology, and it is easy to incorporate many other techniques
- Easy to transfer from a meaningful abstract representation to a coded implementation
- Low processor overhead; well suited to domains where execution time is shared between modules or subsystems. Only the code for the current state need be executed, and perhaps a small amount of logic to determine the current state.
- Easy determination of the ability to reach a state, when represented in an abstract form, it is immediately obvious whether a state is achievable from another state, and what is required to achieve the state

1.4.3. Disadvantages of FSM

- The predictable nature of deterministic FSMs can be unwanted in some domains such as computer games (solution may be non-deterministic FSM).

- Larger systems implemented using a FSM can be difficult to manage and maintain without a well thought out design. The state transitions can cause a fair degree of "spaghetti- factor" when trying to follow the line of execution
- Not suited to all problem domains, should only be used when a systems behavior can be decomposed into separate states with well defined conditions for state transitions. This means that all states, transitions and conditions need to be known up front and be well defined
- The conditions for state transitions are ridged, meaning they are fixed (this can be over come by using a Fuzzy State Machine (FuSM).

1.4.4. States : Some Concepts

In computer science and automata theory, a state is a unique configuration of information in a program or machine. It is a concept that occasionally extends into some forms of systems programming such as lexers and parsers.

Whether the automaton in question is a finite state machine, a pushdown automaton or a full-fledged Turing machine, a state is a particular set of instructions which will be executed in response to the machine's input. The state can be thought of as analogous to a practical computer's main memory. The behavior of the system is a function of,

(a) The definition of the automaton
(b) The input and
(c) The current state.

- **Compatible states** are states in a state machine which do not conflict for any input values. Thus for every input, both states must have the same output, and both states must have the same successor (or unspecified successors) or both must not change. Compatible states are redundant if occurring in the same state machine.
- **Equivalent states** are states in a state machine which, for every possible input sequence, the same output sequence will be produced - no matter which state is the initial state.
- **Distinguishable states** are states in a state machine which have at least one input sequence which causes different output sequences - no matter which state is the initial state.

A state stores information about the past, i.e. it reflects the input changes from the system start to the present moment. A transition indicates a state change and is described by a condition that would need to be fulfilled to enable the transition.

An action is a description of an activity that is to be performed at a given moment. There are several action types:

- **Entry action:** It is performed when entering the state
- **Exit action:** It is performed when exiting the state
- **Input action:** It is performed depending on present state and input conditions
- **Transition action:** It is performed when performing a certain transition

FSM can be represented using a state diagram (or state transition diagram) as in table 1.1. Besides this, several state transition table types are used. The most common representation is shown below: the combination of current state (B) and condition (Y) shows the next state (C). The complete actions information can be added only using footnotes. An FSM definition including the full actions information is possible using state tables.

State transition table			
Current State/Condition	State A	State B	State C
Condition X	...	...	...
Condition Y	...	State C	...
Condition Z	...	...	...

table 1.1: A basic state transition table

1.4.5. Acceptors or Recognizers

There are two types of finite state machine, Acceptors/recognizers and transducers. This paper solely concerned with acceptors or recognizers thus it needs to be discussed elaborately.

Acceptors and recognizers (also sequence detectors) produce a binary output, saying either yes or no to answer whether the input is accepted by the machine

or not. All states of the FSM are said to be either accepting or not accepting. At the time when all input is processed, if the current state is an accepting state, the input is accepted; otherwise it is rejected.

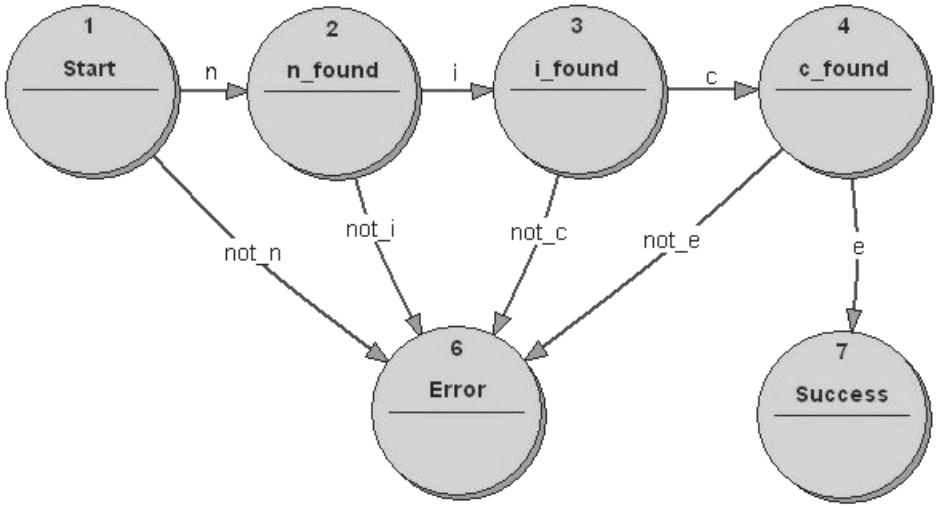
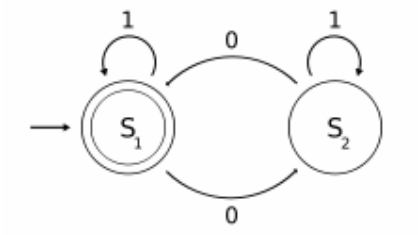


figure 1.2: FSM example of a parser recognizing the world "nice"

As a rule the input are symbols (characters); actions are not used. The example in figure 1.2 shows a finite state machine which accepts the word "nice", in this FSM the only accepting state is number 7.

The machine can also be described as defining a language, which would contain every word accepted by the machine but none of the rejected ones; we say then that the language is accepted by the machine. By definition, the languages accepted by FSMs are the regular languages - that is, a language is regular if there is some FSM that accepts it (cf. Kleene's Theorem).

The start state is usually shown drawn with an arrow "pointing at it from nowhere" (Sipser (2006) p.34). An accept state (sometimes referred to as an accepting state) is a state at which the machine has successfully performed its procedure. It is usually represented by a double circle.



An example of an accepting state appears on the left in this diagram of a deterministic finite automaton which determines if the binary input contains an even number of 0s. S1 (which is also the start state) indicates the state at which an even number of 0s has been input and is therefore defined as an accepting state.

1.4.6. Mathematical Model

Depending on the type there are several definitions. An acceptor finite state machine is a quintuple (Σ,S,s0,δ,F), where:

- Σ is the input alphabet (a finite non empty set of symbols).
- S is a finite non empty set of states.
- s0 is an initial state, an element of S. In a Nondeterministic finite state machine, s0 is a set of initial states.
- δ is the state transition function:.
- F is the set of final states, a (possibly empty) subset of S.

1.5. Edit Distance

1.5.1. Edit Distance

In information theory and computer science, the edit distance between two strings of characters is the number of operations required to transform one of them into the other. There are several different algorithms to define or calculate this metric. In many applications, it is necessary to determine the similarity of two strings. A widely-used notion of string similarity is the edit distance: the minimum number of insertions, deletions, and substitutions required to transform one string into the other.

Edit distance was introduced by Levenshtein, who generalized this concept with multiple edit operations, but did not include transpositions in the set of basic operations. Noticeable point is a transposition error can be considered as a substitution error with length 2.

1.5.2. Levenshtein distance

In information theory and computer science, the Levenshtein distance is a string metric which is one way to measure edit distance. The Levenshtein distance between two strings is given by the minimum number of operations needed to transform one string into the other, where an operation is an insertion, deletion, or substitution of a single character. It is named after Vladimir Levenshtein, who considered this distance in 1965. It is useful in applications that need to determine how similar two strings are, such as spell checkers.

For example, the Levenshtein distance between "kitten" and "sitting" is 3, since these three edits change one into the other, and there is no way to do it with fewer than three edits:

1. kitten → sitten (substitution of 'k' for 's')
2. sitten → sittin (substitution of 'e' for 'i')
3. sittin → sitting (insert 'g' at the end)

It can be considered a generalization of the Hamming distance, which is used for strings of the same length and only considers substitution edits. There are also further generalizations of the Levenshtein distance that consider exchanging two characters as an operation, like in the Damerau-Levenshtein distance algorithm.

This is also known as classic edit distance and the method described in this paper uses this edit distance while ranking its outputs.

1.5.3. Damerau-Levenshtein distance

Damerau-Levenshtein distance is a string metric for measuring edit distance in information theory and computer science. Like Levenshtein distance, it finds the difference between two strings by giving the minimum number of operations needed to transform one string into the other where an operation is an insertion, deletion, or substitution of a single character.

Unlike Levenshtein distance, it counts transposition as a single edit operation, rather than two. Damerau-Levenshtein distance is therefore equal to the minimal number of insertions, deletions, substitutions and transpositions needed to transform one string into the other.

Damerau in his seminal paper not only distinguished these four edit operations but also stated that they correspond to more than 80% of all human misspellings. It is worth noting that Damerau concentrated on single-character misspellings.

1.5.4. Levenshtein Edit Distance Mapping

Levenshtein distance (LD) is a measure of the similarity between two strings, which we will refer to as the source string (s) and the target string (t). The distance is the number of deletions, insertions, or substitutions required to transform s into t.

A commonly-used bottom-up dynamic programming algorithm for computing the Levenshtein distance involves the use of an $(n + 1) \times (m + 1)$ matrix, where n and m are the lengths of the two strings. Here is pseudocode for a function LevenshteinDistance that takes two strings, s of length m, and t of length n, and computes the Levenshtein distance between them:

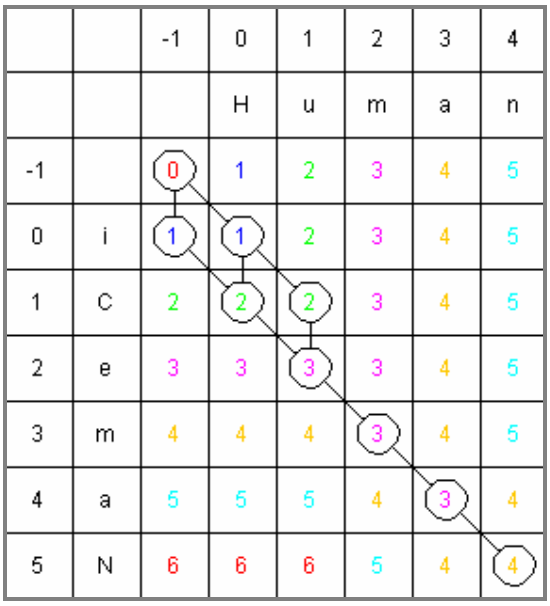
```
int LevenshteinDistance(char s[1..m], char t[1..n])
   // d is a table with m+1 rows and n+1 columns
   declare int d[0..m, 0..n]

   for i from 0 to m
       d[i, 0] := i
   for j from 1 to n
       d[0, j] := j

   for i from 1 to m
       for j from 1 to n
           if s[i] = t[j] then cost := 0
                          else cost := 1
           d[i, j] := minimum(
                                d[i-1, j] + 1,     // deletion
                                d[i, j-1] + 1,     // insertion
                                d[i-1, j-1] + cost   // substitution
                            )

   Return d[m, n]
```

This section shows how the Levenshtein distance is computed when the source string is "iCemaN" and the target string is "Human". The figure also traces the shortest path.



		-1	0	1	2	3	4
			H	u	m	a	n
-1		0	1	2	3	4	5
0	i	1	1	2	3	4	5
1	C	2	2	2	3	4	5
2	e	3	3	3	3	4	5
3	m	4	4	4	3	4	5
4	a	5	5	5	4	3	4
5	N	6	6	6	5	4	4

figure 1.3: Levenshtein distance between “iCemaN” and “Human”

Section 2: The FSR And The Spelling Checker

Section 2: The Spell Checker

2.1. Error Tolerant Finite State Mapping

We can define error-tolerant finite state recognizer as a high error tolerant recognizer of a specific string set supported by the underlying finite state representation of that specific string set (Oflazer, 1996). The important thing is to generate the finite state machine to recognize only the specific string set (All the Entries of Bangla Dictionary) only and nothing else. The term error-tolerance in the other hand requires an error metric for measuring how much the two string deviate from each other. Transposition is rather uncommon in Bangla text and thus Levenshtein distance (accessible at: http://en.wikipedia.org/wiki/Levenshtein_ distance) is the measure of the deviation of two strings.

কর	বল	ধর	খা	যা
করিয়েছিলেন	বলিয়েছিলেন	ধরিয়েছিলেন	খাইয়েছিলেন	
করেছিলেন	বলেছিলেন	ধরেছিলেন	খেয়েছিলেন	গিয়েছিলেন
করছিলেন	বলছিলেন	ধরছিলেন	খাচ্ছিলেন	যাচ্ছিলেন
করিয়েছেন	বলিয়েছেন	ধরিয়েছেন	খাইয়েছেন	গিয়েছেন
করেছিলে	বলেছিলে	ধরেছিলে	খাইয়েছিলে	গিয়েছিলে
করছিলে	বলছিলে	ধরছিলে	খাচ্ছিলে	যাচ্ছিলে
করছেন	বলছেন	ধরছেন	খাচ্ছেন	যাচ্ছেন
করেছেন	বলেছেন	ধরেছেন	খেয়েছেন	গিয়েছেন
করিয়েছিলে	বলিয়েছিলে	ধরিয়েছিলে	খাইয়েছিলে	গিয়েছিলে
করেছিল	বলেছিল	ধরেছিল	খাচ্ছিল	যাচ্ছিল
করিয়েছিল	বলিয়েছিল	ধরিয়েছিল	খাইয়েছিল	গিয়েছিল
করেছে	বলেছে	ধরেছে	খেয়েছে	গেছে
করছে	বলছে	ধরছে	খাচ্ছে	যাচ্ছে
করছ	বলছ	ধরছ	খাচ্ছ	যাচ্ছ
করল	বলল	ধরল	খেল	গেল
করিয়েছে	বলিয়েছে	ধরিয়েছে	খাইয়েছে	গিয়েছে

table 2.1: Some Bangla Verb Roots and Some of The Inflected Forms

For the demonstration purpose we will consider only the currently recognized set at table 2.1 by our working prototype. Figure 2.1 explains the underlying finite state recognizer's principal by mapping some of the roots and its inflections. One thing should be noted is that only the existing strings are recognized and nothing else, even though there are other valid strings that are not in the list. This is the hand-coded, rather optimized state map.

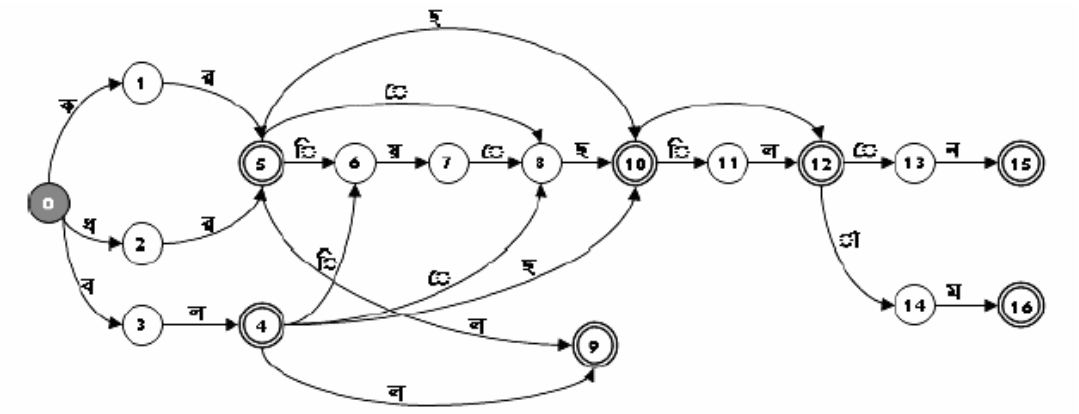


figure 2.1: Finite State Recognizer to Recognize Strings from Table 2.1

The state table can produce by any of the commonly used finite state machine tools such as Xerox FST tool (accessible at: www.xrce.xerox.com/competencies/content-analysis/fssoft/docs/fst-97/xfst97.html) and AT&T FSM library (accessible at: http://www.research.att.com/~fsmtools/fsm/).

The mapping can be extended for every transition representing a property, a feature not required for our implementation at present. For this particular case study, we have used only 10 root words and a handful of inflections, but it is enough to test the expected improvements made to the approach to be used as a non-word spell checker.

The extensive mapping is not used but kept as a future extension to the spell checker to have the ability to work on real-word errors. The data structure for such implementation may vary from the one we are using but concept remains somewhat the same.

2.2. The Algorithm

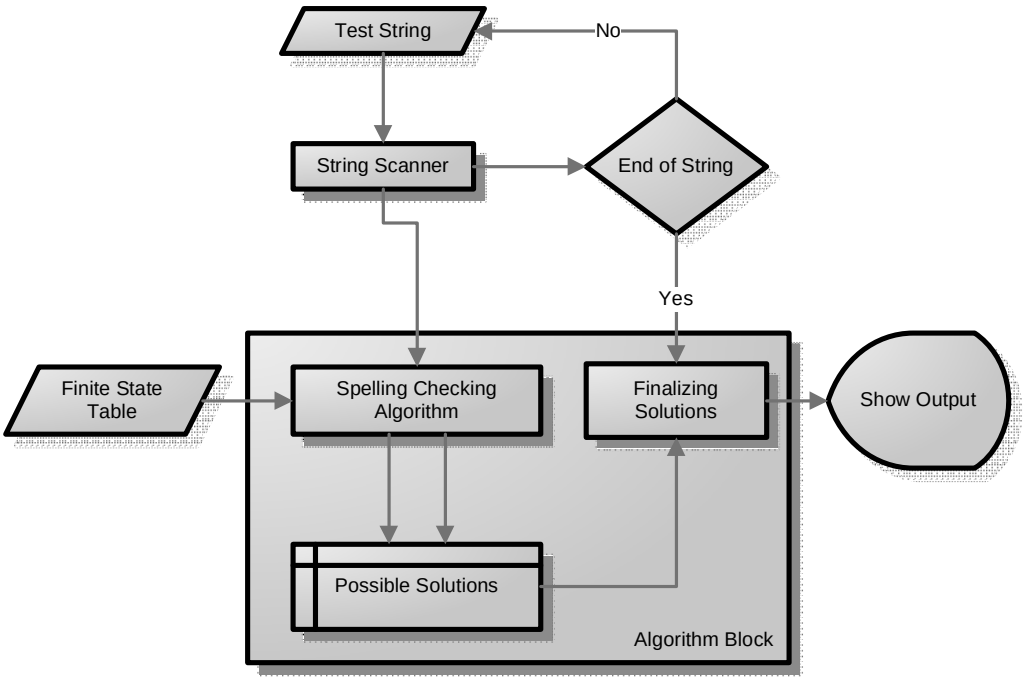


figure 2.2: Spelling Checker System Block Diagram

The spelling checker being developed deals with only isolated non-word errors. The algorithm has been optimized on the basis of statistics to handle the most probable errors efficiently and yet leave the scope to handle the more unusual cases. The algorithm handles substitution and insertion errors somewhat in a similar manner. It uses the look-ahead character to map the next move. On the other hand, deletion error is handled through the current character.

Type of error	Percentage
Substitution	66.90
Deletion	17.87
Insertion	9.60
Transposition	5.63

table 2.2: Rate of Different Errors to Occur in Bengla Text

One advantage in checking spelling using this algorithm is that at every encounter of an erroneous character, it saves the current state and thus uses it with later a character which reduces the scope of misdetection in a multi-character error situation. At the same time it does not leave any possibility unexplored and often produces "absurd" results, but if we take a closer look, the absurd strings turn out to

be probable solutions in terms of string pattern similarity. The list of types of Bangla error found in table 2.2 and the rate of their occurrences are the main motivations behind the spell checking process (Chaudhuri, 2001).

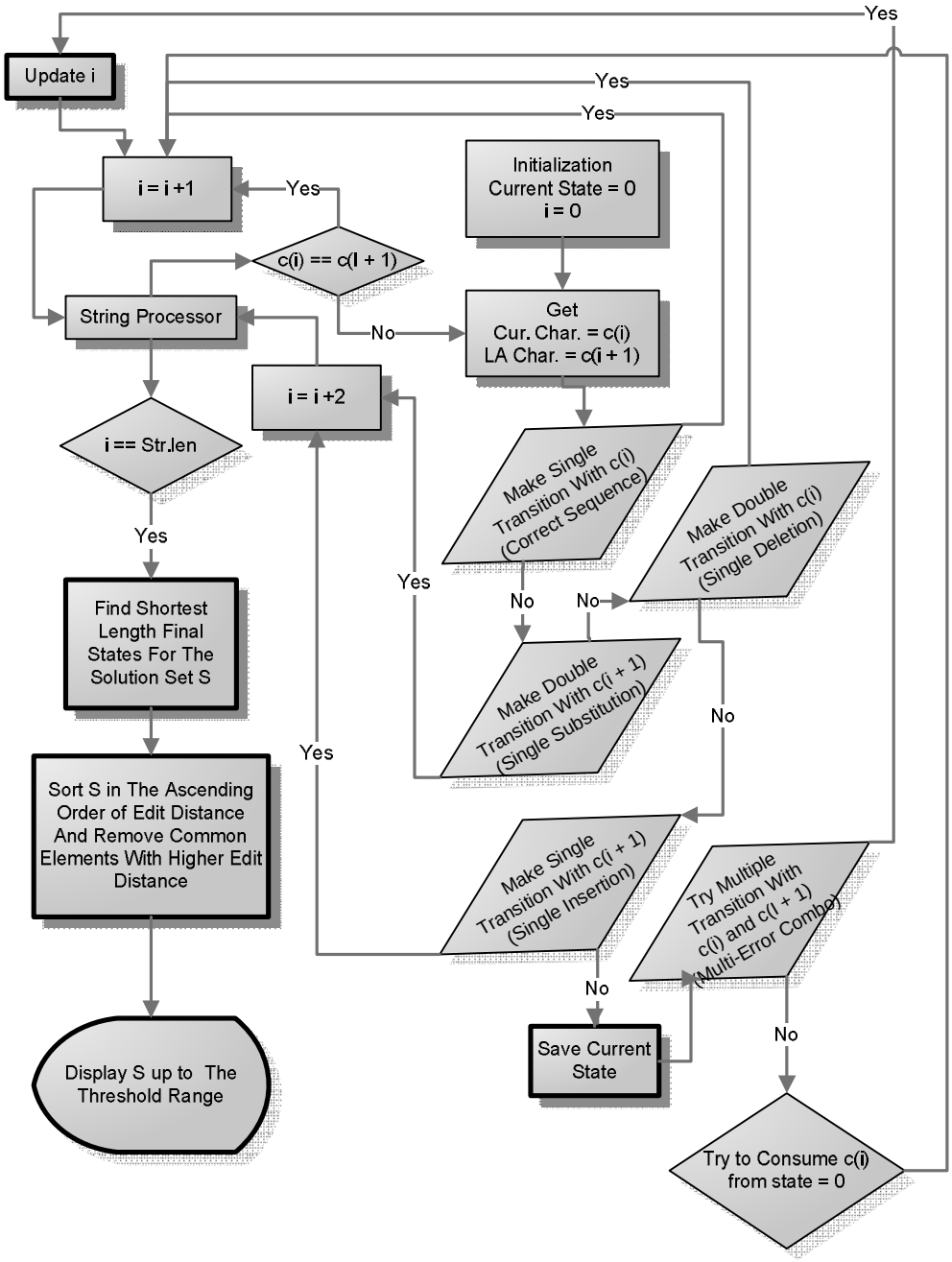


figure 2.3: Process Flow Diagram of The Algorithm

2.3. Implementation

We have used JAVA™ SDK v1.5.0 build 06 to implement our spell checker. While the state tables are meant to be automatically generated from a set of finite state rules, the tables used in the prototype implementation have been hard coded. The algorithm gets the input from the String Scanner, and then uses the Finite-State Recognizer Table to update the set of possible solutions. As the algorithm uses all the possibilities to reach the solution, sometimes our algorithm produces the same string pattern with different edit distance. The finalizing selection process addresses this problem by selecting the final solution set.

2.3.1. Error Detection

The algorithm works on the basis of error occurrence statistics. As suggested by (Chaudhuri, 2001) that single character error is the highest among all Bangla text errors (41.36%) thus it has been handled exclusively. More over the expectation of the presence of a particular type of errors in table 3 also taken under consideration. After the initialization process, we checked if each character occurrence can be consumed by a single transition. If so then we are scanning a correct word. We have a basic check in the main block for repetition by putting an error value in our State-Table for every state. At each failure of such the sequence to assume possible solution is as follows,

a) We checked for Substitution Error (Possible Expectancy 66.90%) by trying to consume the look-ahead character in two transitions. If possible then we switch our next test character to be the immediate next character of the look-ahead character. The recorded edit distance of this move is 1.

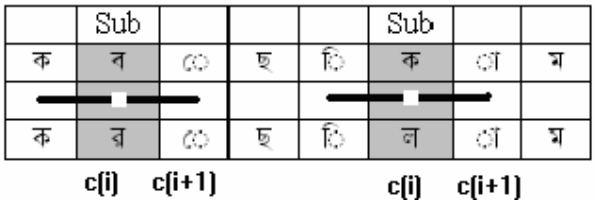


figure 2.4: Handling Substitution Error

b) If we failed in section (a), we check Deletion Error (Possible Expectancy 17.87%) by trying to consume the current character by two transitions. If

possible then we switch our next test character to be the look-ahead character. The recorded edit distance for this move is 1.

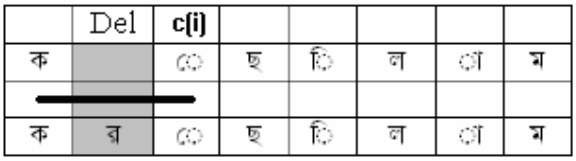


figure 2.5: Handling Deletion Error

c) If we failed in section (b), we check Insertion Error (Possible Expectancy 9.60%) by trying to consume the look-ahead character by a single transition. If possible then we switch our next test character to the look-ahead character. The recorded edit distance for this move is 1.

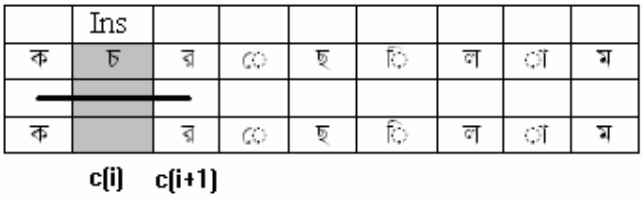


figure 2.6: Handling Insertion Error

d) If we failed in section (c), we first save the current state for later use and check Multiple Character Deletion Error by trying to consume the current character by a multiple transitions. If possible then we switch our next test character to be the look-ahead character. The recorded edit distance for this move is the sum of the number of transitions and 1 and it is updated to the saved state as well.

e) If we failed in section (d), then we assume a combinational multiple errors and try to consume the look-ahead character in multiple transitions. If possible then we switch our next test character to be the immediate next character to the look-ahead character. The recorded edit distance for this move is the sum of the number of transitions and 1.

f) If all attempts are failed we make a bold assumption that all the characters before the current character are insertions and make a multi-transition from the start state to consume the current character and the try the next

character. The recorded edit distance for this move is the sum of the number of transitions and the position of current character.

One thing that should be noticed is, we save the current state just after failing with single error checking. This eventually will help us to lead to the correct solution as multiple moves with the next test character will be made from that state. All these checks will be made with all the elements of the solution set thus no possibilities will be left off and a synchronous edit distance update is made to keep things in order.

Another significant benefit from this approach is the reduced number of pattern matching. The algorithm will make deterministic moves rather than trying a blind guess. This reduces the processing time needed for each check and increases the speed.

2.3.2. Suggestion Generation

After checking the whole string a finalizing method is used. In this step the possible output set is checked, sorted and selected by threshold matching. All the possibilities are finalized by making them reach a final state by the minimum number of moves and it is tried in both forward and reverse direction. Due to the mix of singular and multiple moves the prototype often produces same patterns with different edit distance. Similar patterns with higher edit distance are removed after sorting the list. Then the elements having an edit distance higher than a predefined threshold value are truncated. Thus the result is ready.

2.3.3. Suggestion Ranking

The ranking is the most important feature of a spelling checker. The optimized the ranking method is the better and smaller in number the suggestion would be. In case of an edit distance based ranking method is used in a system where huge numbers of results are produced with many of those having the same edit distance, it is hard for a spelling checker to rank them.

The proposed system for a starter produces small number of suggestions and it is the least probable to have too many suggestions with the same edit distance.

More over the threshold used is not some specific edit distance but the number of top ranking results. Standard for this number should not be more than 10.

2.4. Performance and Evaluation

The evaluation metric used for testing the performance is based on the (Kukich, 1992) parameters for isolated-word error correction:

- Lexicon size;
- Test set size;
- Correction accuracy for single error misspellings;
- Correction accuracy for multi-error misspellings; and
- Type of errors handled (phonetic, typographical, OCR generated etc.).

These metrics have been used by another notable effort in Bangla spelling checker as well (Bhatt, Choudhury, Sarkar and Basu 2005).

There were 3 files in the test set, each with 97 words. One file contained all correct strings, another contained strings with single error of any of the possible types, and the third file contained string with multiple, ranging from 2 to 9, character errors. The results, with the actual clock time, are shown in Table 3.

The lexicon used in this study does not contain the actual words, rather consists of a state table with 50 states for the 97 strings used in the test case. The results show 92% accuracy for single character error in different positions. The algorithm fails to find the suggestion in cases where the transposition error occurs at either end of the word, and that accounts for much of the remaining 8% in our results.

In case of multiple character errors, the accuracy is approximately 70%. It should be noted that some of the input words were randomly generated nonsense words, which often led to the algorithm producing absurd suggestions in return.

System Configuration	Error Type	Words	Worst time (ms)		Best Time (ms)		Average Time (ms)	
			T1	T2	T1	T2	T1	T2
Intel® Pentium® 4/ 2.80 GHz/ 512 MB RAM	No Error	97	109	110	31	31	70	70.5
	Single Error	97	126	125	46	46	86	85.5
	Multiple Error	97	188	188	78	77	133	132.5
Intel® Pentium® 3/ 800 MHz/ 128 MB RAM	No Errors	97	231	140	70	70	150.5	105
	Single Error	97	230	220	100	110	165	165
	Multiple Error	97	401	371	230	240	315.5	305.5

table 2.3: Actual Test Results on Two Different Systems

An attempt was made to check the suggestion generation capabilities of the algorithm for any word, including those that may never occur in Bangla. One important result is the very small incremental time needed to find single-error misspellings over the correct ones. The state transition approach keeps the transition time somewhat the same and that should be regarded as a significant improvement in the performance of the spell checker.

2.5. Future Improvements

One notable improvement would be to introduce transposition errors as a single edit distance errors. If that is done, a simple check of single transition possibility based on the current character and the look-ahead character will yield a possible suggestion. This type of errors is not impossible to correct with the current algorithm but it is considered as a double substitution and thus higher frequency of such error in a text will slow the overall process.

A faster and optimized state table generation can further improve the performance. In the Error-tolerant Finite State Recognition with Applications to Morphological Analysis and Spelling (Oflazer, 1996), proposed an algorithm for such a task and yet a more optimized approach can be obtained. Improved data structure like the Recursive Transition Network (RTN) is well suited for such a task.

Extending this algorithm to handle real-word errors can be achieved by augmenting the information kept for the mapping of the states, which is currently used for state status determination only. Memory requirement is another concern in the current implementation, which can be significantly improved by using different representations such as node-edge representation for the state table. The transition speed may also be improved by using a DBMS approach while searching.

Conclusion

This paper presents a finite state machine based spelling checker for Bangla that shows good promise in terms of suggestion generation and runtime performance. The method described optimizes its performance by using statistical information on the error patterns of Bangla text. While it is quite possible to use this method on other languages without such statistics, the performance will definitely suffer. The approach leaves much space for improvement and may be able to have practical application with devices with low memory. The key feature is there is no dictionary.

References

Naushad UzZaman and Mumit Khan. 2006. A Comprehensive Bangla Spelling Checker. In the *Proceeding of International Conference on Computer Processing on Bangla (ICCPB- 2006)*, Dhaka, Bangladesh, 17 February.

P. Kundu and B.B. Chaudhuri. 1999. Error Pattern in Bangla Text. *International Journal*
of Dravidian Linguistics, 28(2).

B. B. Chaudhuri. 2001. Reversed word dictionary and phonetically similar word grouping based spell-checker to Bangla text. *LESAL Workshop*, Mumbai.

Arif Billah Al-Mahmud Abdullah and Ashfaq Rahman. 2004. A Different Approach in Spell Checking for South Asian Languages. In the *Proceeding 2nd International Conference on Information Technology for Applications (ICITA)*, China.

Kermal Oflazer. 1996. Error-tolerant Finitestate Recognition with applications to Morphological Analysis and Spelling correction. *computational Linguistics*, Volume 22, page: 73 – 89.

Sebastian Deorowicz and Marcin G. Ciura. 2005. Correcting Spelling Errors by Modeling their Causes. *International Journal of Applied Mathematics and Computer Science.* Vol. 15, No. 2, 275–285.

K. Kukich, 1992. Techniques for automatically correcting words in text. *ACM Computing Surveys*, 24(4):377--439.

Naushad UzZaman and Mumit Khan. 2005. A Double Metaphone Encoding for Bangla and its Application in Spelling Checker. *IEEE International Conference on Natural Language Processing and Knowledge Engineering*, Wuhan, China.

Naushad UzZaman and Mumit Khan. 2004. A Bangla Phonetic Encoding for Better Spelling Suggestions. In the *Proceeding of 7th International Conference on Computer and Information Technology (ICCIT 2004)*, Dhaka, Bangladesh, December.

Aniruddh Bhatt, Monojit Choudhury, Sudeshna Sarkar and Anupam Basu. 2005. Exploring the Limits of Spellcheckers: A comparative Study in Bangla and English. *The Second Symposium on Indian Morphology, Phonology and Language Engineering (SIMPLE'05). Published by CIIL Mysore,* pages. 60 - 65, Kharagpur, INDIA.