Thesis paper submitted to the Department of Computer Science and Engineering, Shahjalal University of Science & Technology, in partial fulfillment of the requirement for the degree of Bachelor of Science (Engineering).

Thesis Supervisor	**Thesis Committee**
Abu Naser	Dr. Shahidur Rahman
Lecturer,	Head and Professor,
Department of Computer Science and Engineering,	Department of Computer Science and Engineering,
Shahjalal University of Science & Technology,	Shahjalal University of Science & Technology,
Sylhet - 3114	Sylhet - 3114

5th May 2014

Auto Correction, Auto Completion and Search Suggestion for Bangla Search Engine

Department of Computer Science and Engineering
Shahjalal University of Science and Technology
Sylhet – 3114

Report for CSE 404

Submitted By
Syed Shahriar Manjur, 2009331001
Imtiaz Shakil Siddique, 2009331006
Nafis Ahmed, 2009331042

Acknowledgement

We want to thank several people for helping us on our research and sharing their useful thoughts and ideas on developing Bangla Search Suggestion and Auto Completion. We are very grateful to our thesis supervisor Abu Naser, who helped us throughout the entire thesis. We would also like to thank Md. Saber Ismail for reviewing our thesis and also Md. Ranju Sir for providing us with huge collection of Bangla corpus data. We also want to thank the whole Pipilika team for providing us with their query log and suggesting us on some important features regarding our thesis.

Abstract

Search suggestion and auto completion is an essential part of every modern day search engines like Google, Bing, Yandex etc. Comparatively it is a new feature even for Giant search engine like Google. But its usefulness can't be overlooked as it is widely accepted by every users around the globe.

Our Thesis report demonstrates a practical solution for Query Suggestion and Auto Completion. We are able to provide Suggestions based on Query Log as well as Documents which are available throughout the internet.

Table of Contents

1. Introduction

Search engine has come a long way since 1998 when Google first launched as a search result provider on the internet. Now a days it has become an intrigated part in our day to day life. Since then many features have been introduced for user's convenience. Among them there is one such feature that has been widely accepted by most people and been implemented on other popular search engines. This feature is called Search Suggestion and Auto Completion. Though it already implemented and being used massively on popular search engines such as Google, Yandex, Bing etc. But still there are no such service for Bangla Search Engine.

Currently in our country we have only one Search Engine, Pipilika which provides Bangla search results. Still the suggestion and auto completion service has yet to be intrigated on Pipilika. Our research aims for the solution on this specific service.

2. Our Goal

Search Suggestion and Auto Completion is a very essential part of any modern search engine. But we have to keep in mind that there are lot of problems that needs to be addressed in order to actually build such an intrigation system.

Our Goal is to provide this Search Suggestion and Auto Completion service that provides relevant answers in reasonable time. Our thesis greatly emphasizes on the run time and the memory constraint while providing such service.

3. Whole System in Brief

Providing Search Suggestion and Auto Completion in Bangla is a huge task which is divided into three major segments. They are:

- **Query Auto Correction**
- **Query Auto Completion**
- **Query Suggestion**

in Bangla.

4. Query Auto Correction Related Problem

Search engine users very frequently makes spelling mistakes, more so when they are searching in a Bangla Search Engine (Pipilika). The query log we got from Pipilika was full with spelling mistakes for which the users didn’t get the results they were looking for. Automatic Spell Correction is a way to fix this issue.

But these are a lot of difficulties when building an Auto Corrector. We have to have a huge dictionary of correct words and compare them with the word in the search query to find out if it’s correct or not. We also have to make a suggestion of the possible correct words if the word in query is not a valid one. Again replacing the error with a correct word does not necessary mean that it was the word the user was looking to write.

Keeping all these in mind we also have to think about the time complexity of the data structure and the algorithm we want to use to solve this problem.

4.1 BK Tree

BK Trees, or Burkhard-Keller Trees are a tree-based data structure engineered for quickly finding near-matches to a string, for example, as used by a spelling checker, or when doing a 'fuzzy' search for a term. The aim is to return, for example, "seek" and "peek" if I search for "aeek". What makes BK-Trees so cool is that they take a problem, which has no obvious solution besides brute-force search, and present a simple and elegant method for speeding up searches substantially.

BK-Trees were first proposed by Burkhard and Keller in 1973, in their paper "Some approaches to best match file searching". The only copy of this online seems to be in the ACM archive, which is subscription only. Further details, however, are provided in the excellent paper "Fast Approximate String Matching in a Dictionary".

Before we can define BK-Trees, we need to define a couple of preliminaries. In order to index and search our dictionary, we need a way to compare strings. The canonical method for this is the Levenshtein Distance, which takes two strings, and returns a number representing the minimum number of insertions, deletions and replacements required to translate one string into the other. Other string functions are also acceptable (for example, one incorporating the concept of transpositions as an atomic operation could be used), as long as they meet the criteria defined below.

Now we can make a particularly useful observation about the Levenshtein Distance: It forms a Metric Space. Put simply, a metric space is any relationship that adheres to three basic criteria:

d(x,y) = 0, then x = y (If the distance between x and y is 0, then x = y)

d(x,y) = d(y,x) (The distance from x to y is the same as the distance from y to x)

d(x,y) + d(y,z) >= d(x,z)

The last of these criteria is called the Triangle Inequality. The Triangle Inequality states that the path from x to z must be no longer than any path that goes through another intermediate point (the path from x to y to z). Look at a triangle, for example: it's not possible to draw a triangle such that it's quicker to get from one point to another by going along two sides than it is by going along the other side.

These three criteria, basic as they are, are all that's required for something such as the Levenshtein Distance to qualify as a Metric Space. Note that this is far more general than, for example, a Euclidian Space - a Euclidian Space is metric, but many Metric Spaces (such as the Levenshtein Distance) are not Euclidian. Now that we know that the Levenshtein Distance (and other similar string distance functions) embodies a Metric Space, we come to the key observation of Burkhard and Keller.

Assume for a moment we have two parameters, query, the string we are using in our search, and n the maximum distance a string can be from query and still be returned. Say we take an arbitrary string, test and compare it to query. Call the resultant distance d. Because we know the triangle inequality holds, all our results must have at most distance d+n and at least distance d-n from test.

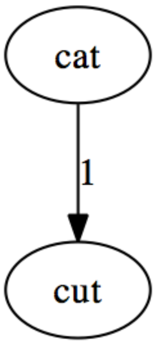
4.2 Construction for BK Tree

Each node has a arbitrary number of children, and each edge has a number corresponding to a Levenshtein distance. All the sub nodes on the edge numbered n have a Levenshtein distance of exactly n to the parent node. So, for example, if we have a tree with parent node "book" and two child nodes "rook" and "nooks", the edge from "book" to "rook" is numbered 1, and the edge from "book" to "nooks" is numbered 2.

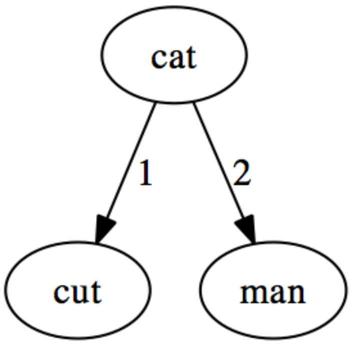
To build the tree from a dictionary, take an arbitrary word and make it the root of your tree. Whenever you want to insert a word, take the Levenshtein distance between your word and the root of the tree, and find the edge with number d(newword,root). Recurs, comparing your query with the child node on that edge, and so on, until there is no child node, at which point you create a new child node and store your new word there. For example, to insert "boon" into the example tree above, we would examine the root, find that d("book", "boon") = 1, and so examine the child on the edge numbered 1, which is the word "rook". We would then calculate the distance d("rook", "boon"), which is 2, and so insert the new word under "rook", with an edge numbered 2.

To query the tree, take the Levenshtein distance from your term to the root, and recursively query every child node numbered between d-n and d+n (inclusive). If the node you are examining is within d of your search term, return it and continue your query.

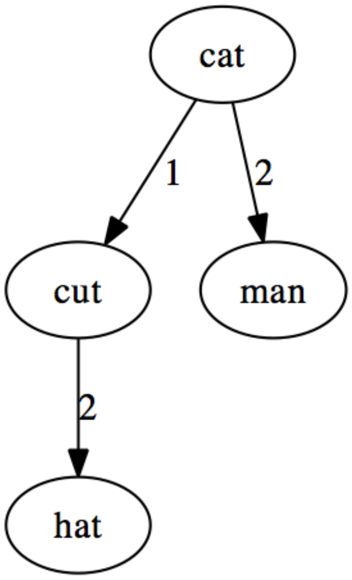
So if we have the following list of words: ["cat", "cut", "hat", "man", "hit"], we will start by creating a “cat” node with no children. To add “cut” we calculate the Levenshtein distance to be one and insert it under the “cat” node.



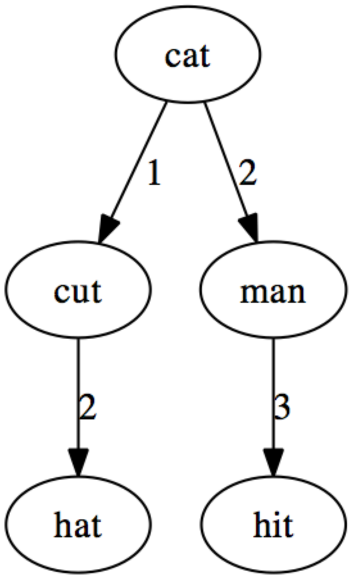
1) Cut is inserted under cat with a branch of length one



2) The Levenshtein distance between “cat” and “man” is two, so “man” is connected with a branch of length two



3) d("hat","cat") = 1, so the insertion operation is done on the "cut" node, and "hat" is connected to "cut" with a branch of length two (d("cut","hat")=2)



4) d("cat","man") = 2, so the insertion operation is done on the "man" node, and "hit" is connected to "man" with a branch of length three (d("man","hit")=3)

The query algorithm is also simple: We find the distance d from the query word to the root node. If this is less than the maximum distance we allow, n, we include this word in the result. We then find all the child nodes connected by branches of length (d-n) ≤ l ≤ (d+n) and recursively query these nodes and add to the result. The result is a list of words satisfying d(query_word, word) ≤ n.

4.3 N-Gram Model

In the fields of computational linguistics and probability, an n-gram is a contiguous sequence of n items from a given sequence of text or speech. The items can be phonemes, syllables, letters, words or base pairs according to the application. The n-grams typically are collected from a text or speech corpus.

An n-gram model is a type of probabilistic language model for predicting the next item in such a sequence in the form of a (n-1) order Markov model. n-gram models are now widely used in probability, communication theory, computational linguistics (for instance, statistical natural language processing), computational biology (for instance, biological sequence analysis), and data compression.

4.4 How we are using BK Tree

As for Bangla, we don't have any dictionary, we are creating our own. We have parsed almost 50,000 newspaper and blog links and created a wordlist using them.

Using BK Tree we can find if the word in our query is correct or not and quickly give words that are within some given levenshtein distance from our query word. But it doesn't necessary mean that this word is the actual word the user was trying to write.

What we are doing is that, we are calculating the frequency of each word in our corpus data and generating our suggested list of word sorted by the most used word in our corpus to the least used word.
But this model gives unacceptable results when it is generating the probable correct words using only the frequency.

User Query: আমি জনি
Corrected Query: আমি জন

So we also used N-Gram Model, which calculates from the array of suggested words, which word is more probable than others with respect to the previous words in the user's query.

User Query: আমি জনি
Corrected Query: আমি জানি

Again it is possible to incorrectly write a correct word, which has a valid meaning, but it's incorrect with respect to the previous words in the user's query. For this reason, we have to search our dictionary and calculate each word's probability for valid words also.

User Query: শেখ হাস্না

Corrected Query: শেখ হাসিনা

Sometimes a user inputs a query, in which there is a word that is very rare but correct. In these cases we cannot use our n-gram model to autocorrect this word. What we are doing here is that we are not looking for the most probable word but we are just checking if this word has even a slight probability of being correct. And if it is then we let it remain as it is.

User Query: শেখ হাসিনা ও শেখ

Corrected Query: শেখ হাসিনা ও শেষ (should be শেখ)

4.5 Performance Analysis

Using the BK-Tree and merging the N-gram model with it is very fast with respect to the all the other models available for searching and fixing a wrong word using a Dictionary.

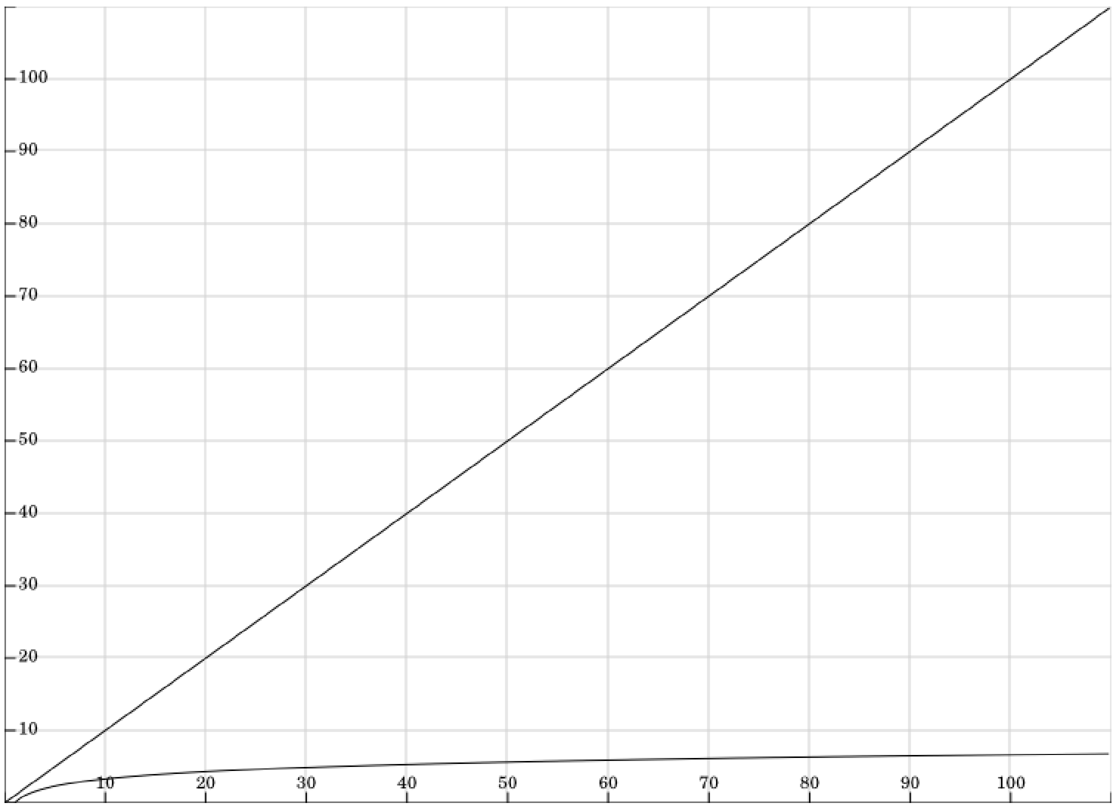


Figure: Comparison between search time of BK-Tree and other known algorithms

We can pre calculate the BK-Tree and it has a complexity of O(N * L^2), where N is the number of words in the Dictionary and L is the maximum length of the words.

Each query has a complexity of

- O(logN * L^2), for the BK-Tree, where N is the number of words in the Dictionary and L is the maximum length of the words.
- O(M*K) where is the number of word in the user's query and K is the size of the suggested correct wordlist.

4.6 Future Plans:

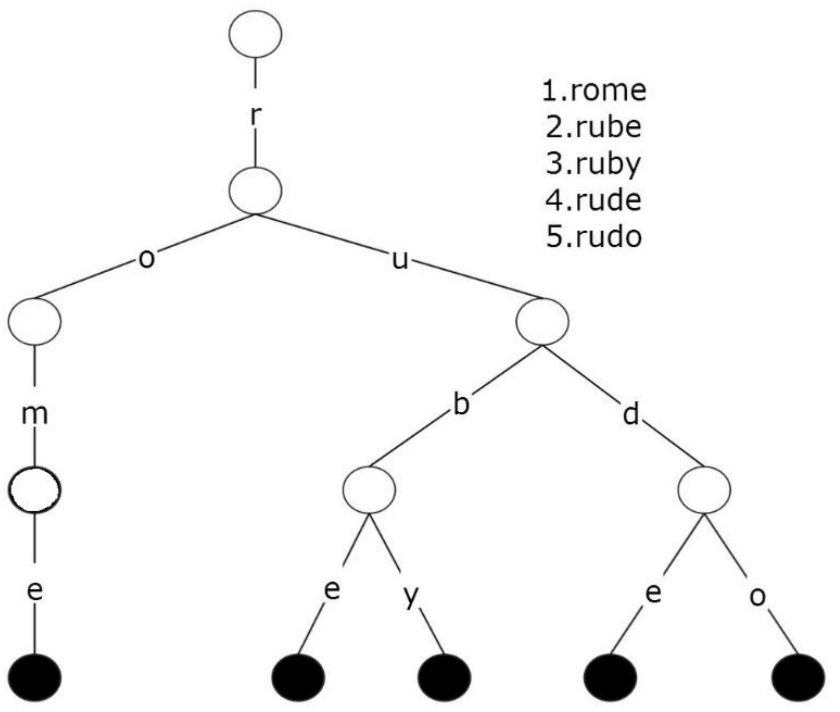
- Calculating the levenshtein distance is somewhat time consuming and it also needs to be calculated in each query. One possible may be to use Levenshtein Automata.
- N-gram model in a reverse order may help us in giving a more accurate query by fixing the previous word in a query with regards to the current word.

5. Query Auto Completion and related problem

- Huge amount of query needs to be stored and processed each day
- Weight of a query changes dynamically from time to time thus the ability to detect such changes
- Automated Suggestion and information retrieval needs to be done almost instantly
- Relevant and accurate information needs to be retrieved

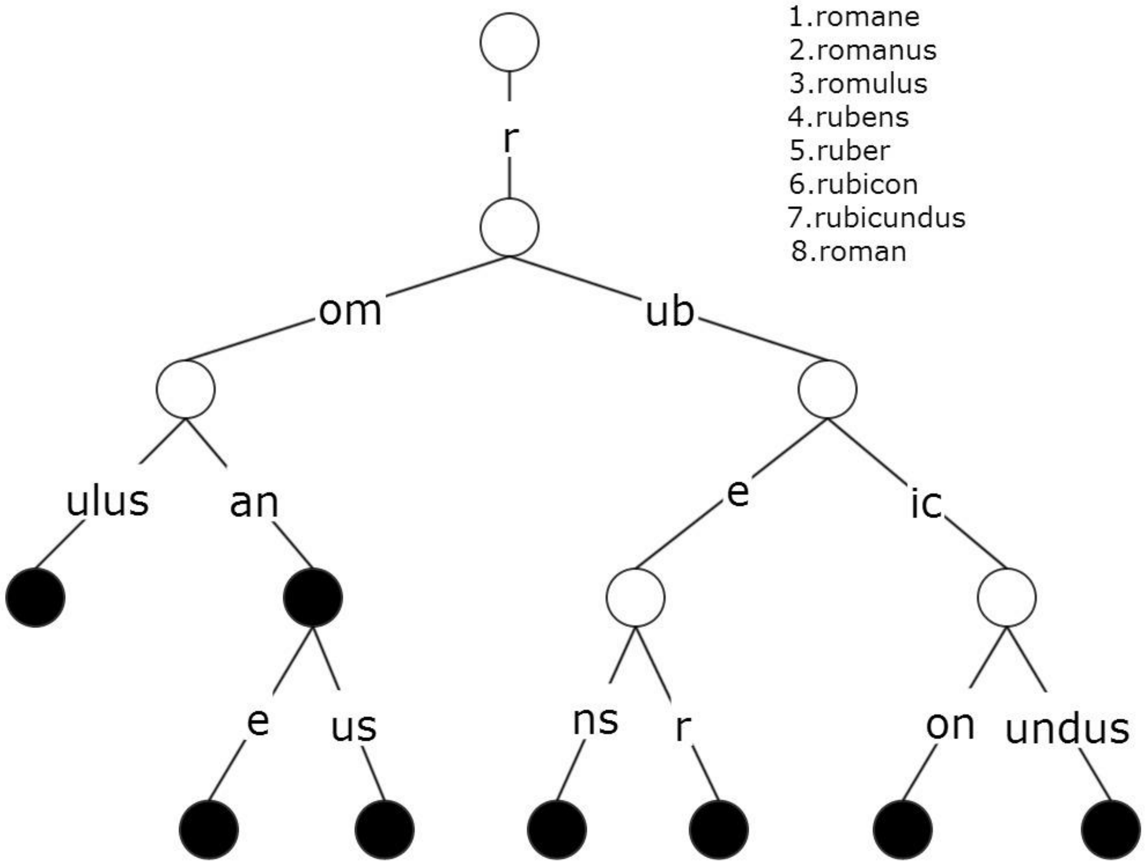
5.1 Compressed Trie: A Possible Solution

In order to understand compressed trie we must introduce trie data structure which is a very ancient data structure
The word trie is an infix of the word "re**trie**val" because the trie can find a single word in a dictionary with only a prefix of the word. The main idea of the trie data structure consists of the following parts:



The trie is a tree where each vertex represents a single word or a prefix. The root represents an empty string (""), the vertexes that are direct sons of the root represent prefixes of length 1, the vertexes that are 2 edges of distance from the root represent prefixes of length 2, the vertexes that are 3 edges of distance from the root represent prefixes of length 3 and so on. In other words, a vertex that are k edges of distance of the root have an associated prefix of length k.
Let **v** and **w** be two vertexes of the trie, and assume that **v** is a direct father of **w**, then **v** must have an associated prefix of**w**.The next figure shows a trie with the words "rome", "rube", "ruby", "rude", "rudo".

In Trie data structure each prefix of a word is stored only once but for each new string it stores nodes exactly the size of the string.Thus Trie has some certain disadvantage as we can see from the figure.It stores too many unnessary nodes which in our case is very fatal.As we have said query sentences are huge, if we store



all the query sentences in a normal trie our memory would run out.
Thus the solution of this problem is Compressed Trie. We will try to simulate compressed trie with the following figure.
Compressed Trie:
A compressed trie is a trie with one additional rule:

1. Each internal node has ≥ 2 children
2. Obtained from standard trie by compressing chains of redundant nodes

So inorder to implement Compressed Trie we must stop using redundant nodes which from a redundant chain.Related information on compressed trie can be found on this link:
http://www.mathcs.emory.edu/~cheung/Courses/323/Syllabus/Text/trie02.html.

5.2 Advantages of using Compressed Trie

1. We can seek out the required information extremely fast, at constant time i.e. O(1) complexity.
2. Prefix matching can be implemented in compressed trie which is in our case absoloutely necessary.
3. The total storage required by compressed trie is extremely efficient when storing large quantity of query sentences as compressed trie compresses number of nodes used.
4. Insert and Delete operation is extremely fast in compressed trie.
5. For Weighted Information Retrieval Compressed Trie can be modified.
6. Compressed Trie allows us to backup the whole tree which is important if we are to intrigate this service in any online search engine as their server is prone to crashing.
7. Number of Suggestions retrieved in a compressed trie can be easily controlled.

5.3 Challenges of implementing Compressed Trie:

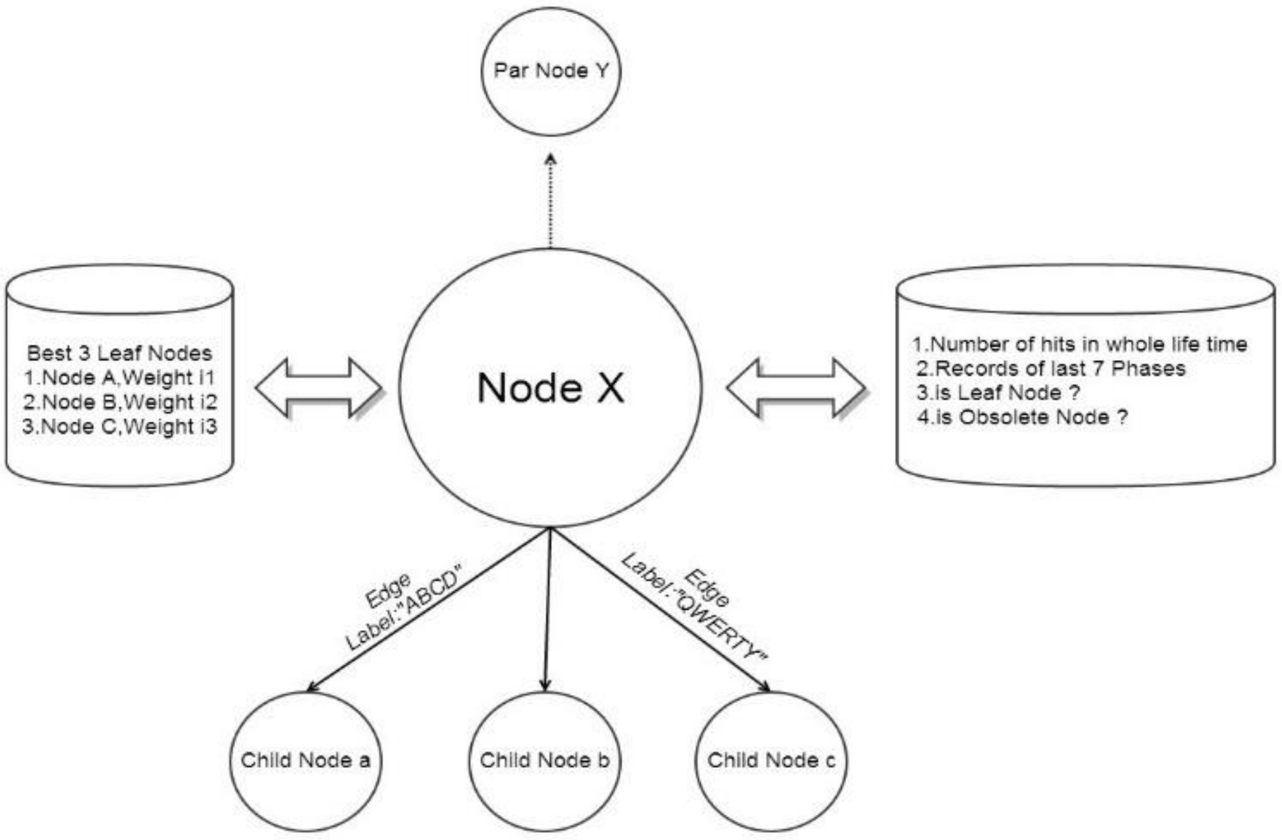
1. How we seek out the most relevant information from Compressed Trie Tree.
2. How to calculate the weight of a query string and implement it on such tree.
3. Weight of a leaf node in a compressed trie tree changes dynamically, How to implement this?
4. What information needs to be stored which gives us all the information we require and all this at minimal storage requirement.

5.4 Proposed Solution and Short Description of Implementation:

Inorder to solve all the required challenges and we have devised an interesting solution.To understand our proposed solution you must know what type of information we are storing in each node in a compressed Trie tree.First of Each node in our Compressed Trie Tree should have some specific required information which are:

- Link to parent Node
- Link to all child
- Global number of hits of this node in its Life time.
- Records(number of hits) of last 7 Phases of this current node.
- Best Weighted 3 or 4 nodes who are child of this current node (immediate or not)
- Current node leaf node or not
- Current node obsolete node or not

Why such information is stored will be discussed in brief shortly. These specific information is enough for our purpose of retrieving information from the Compressed Trie Tree.



5.4.1 Weight Calculation of a Node

Before simulating the information retrieval process we must clearly define how we are defining weights of each node and show some examples.

Case 1.

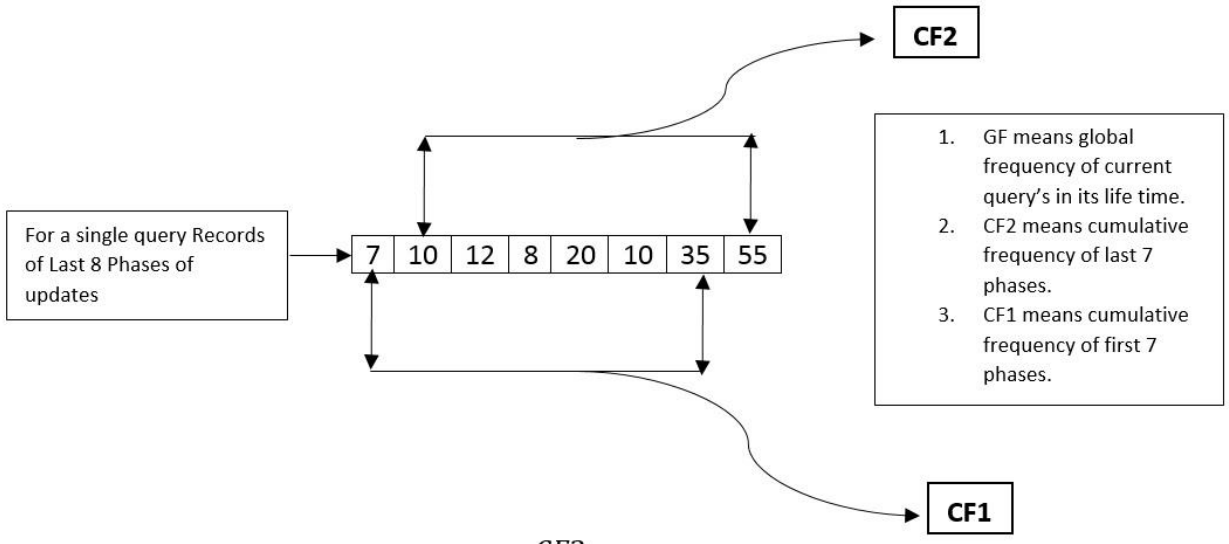
Weight of a specific query can go up and down due to many reasons. One of them is due to change of time.An Useful Example is given below.

Popular Queries in Summer	Popular Queries in Winter	Popular Queries in All Seasons
সুতির পাঞ্জাবী	সোয়েটার এর দাম	আজকের খবর
আইস ক্রিম	হিটার পাবো কোথায়?	বাংলাদেশের দর্শনীয় জায়গা গুলো কি কি?

Case 2.
There are some queries which can gain instantly become very much popular and within a short interval of time can lose its popularity.

Queries that gain and lose popularity in short interval
আইফোন ৫ এর দাম বাংলাদেশে কত?
মেসি ফর্ম এ নেই কেন?
বাংলাদেশ ক্রিকেট দলের খেলার ফলাফল?

By analyzing above requirements we have mathematically defined weight of every single query that is every single leaf node.Suppose we have records of last 8 phases of every single query.We also know the global frequency GF of every single leaf node.



1. GF means global frequency of current query's in its life time.
2. CF2 means cumulative frequency of last 7 phases.
3. CF1 means cumulative frequency of first 7 phases.

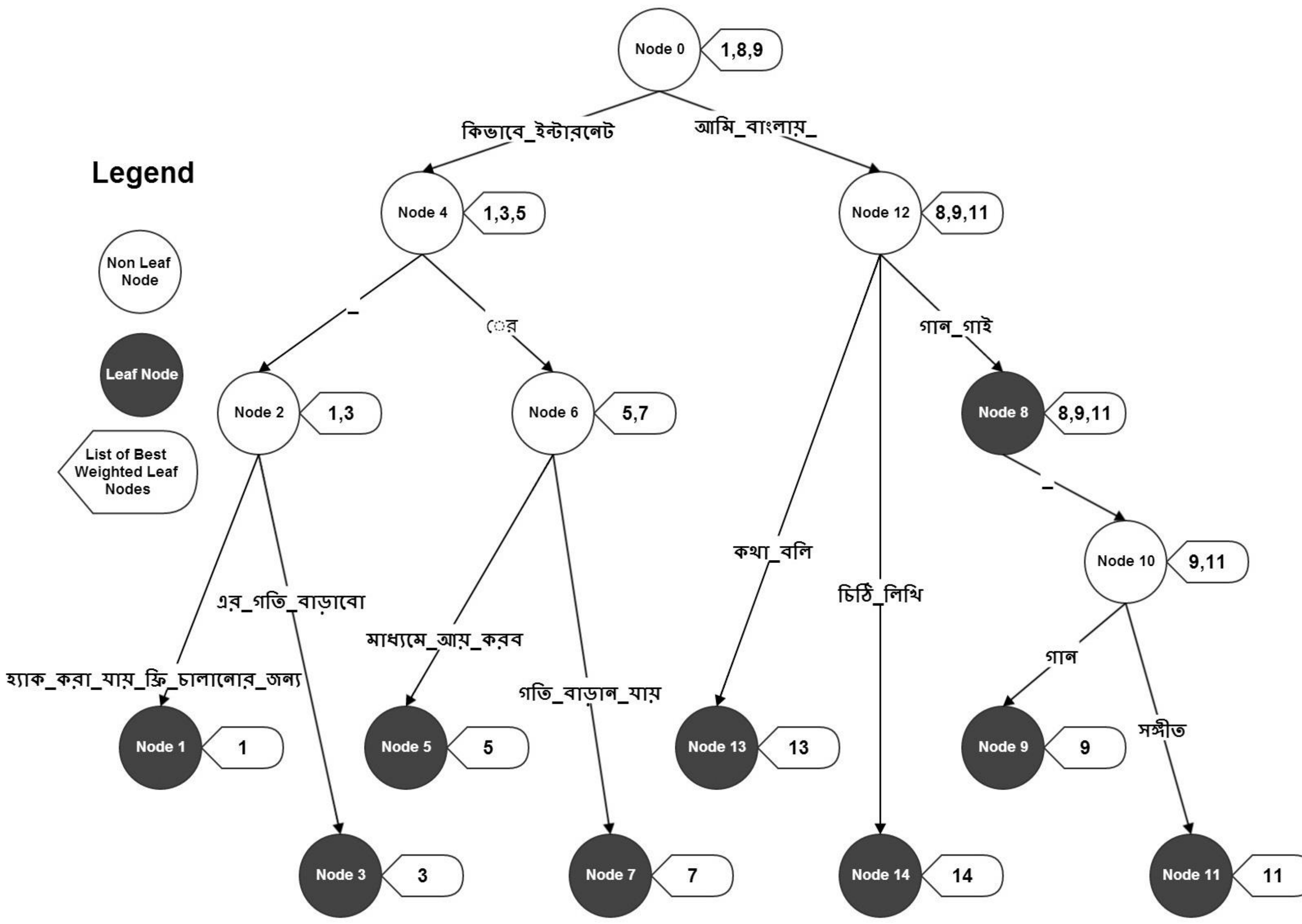
Weight Calculation: $\frac{CF2}{CF1} * GF + \log_2 GF$

5.4.2 Building the Tree

For Each query string we update the whole tree and in the leaf node we update

1. Its global frequency
2. Calculate its weight using the Weight Calculation Formula.
3. Update the phase records i.e. last record is removed while new record is inserted.

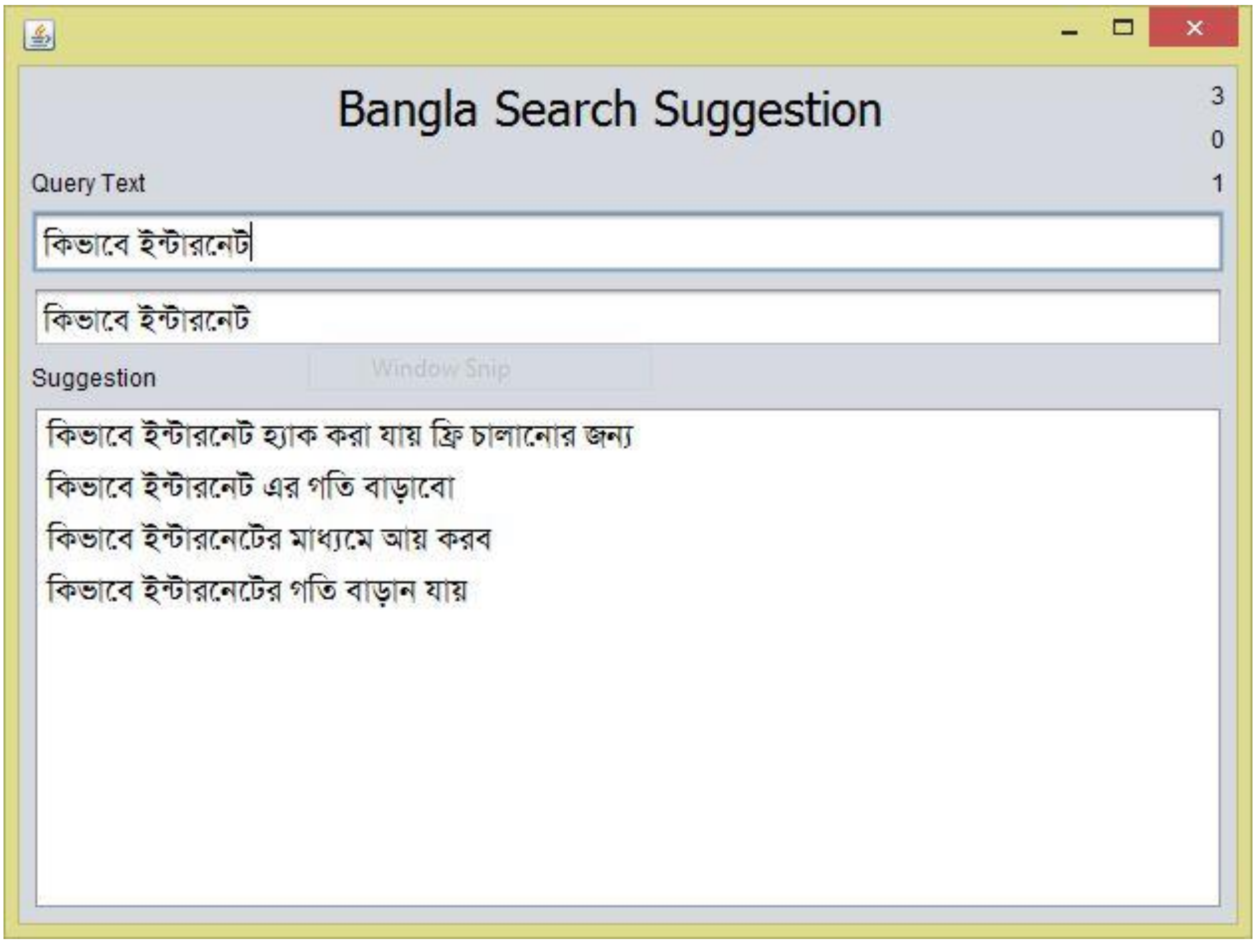
For Every single node other than the leaf node we do all the above things except point 2 and 3.At the same time for a non-leaf node we update the list of best leaf nodes which is its immediate or non-immediate child.The list of best leaf nodes is always sorted according to each leaf node's Weight which is generated at creation of the leaf node.



In the figure we have 8 query sentences.They are given as well as their weight is also given.When we completed updating all this sentences in Compressed Trie the tree looks like above.

Query Sentence	Weight
কিভাবে_ইন্টারনেট_হ্যাক_করা_যায়_ফ্রী_চালানোর_জন্য	৫৫
কিভাবে_ইন্টারনেট_এর_গতি_বাড়াবো	৮
কিভাবে_ইন্টারনেটের_মাধ্যমে_আয়_করব	৬
কিভাবে_ইন্টারনেটের_গতি_বাড়ান_যায়	৪
আমি_বাংলায়_গান_গাই	৫০
আমি_বাংলায়_গান_গাই_গান	৩০
আমি_বাংলায়_গান_গাই_সঙ্গীত	১৫
আমি_বাংলায়_কথা_বলি	১০
আমি_বাংলায়_চিঠি_লিখি	৬

Now suppose User Searched for “কিভাবে ইন্টারনেট” then we will get below output.



Bangla Search Suggestion

Query Text

কিভাবে ইন্টারনেট

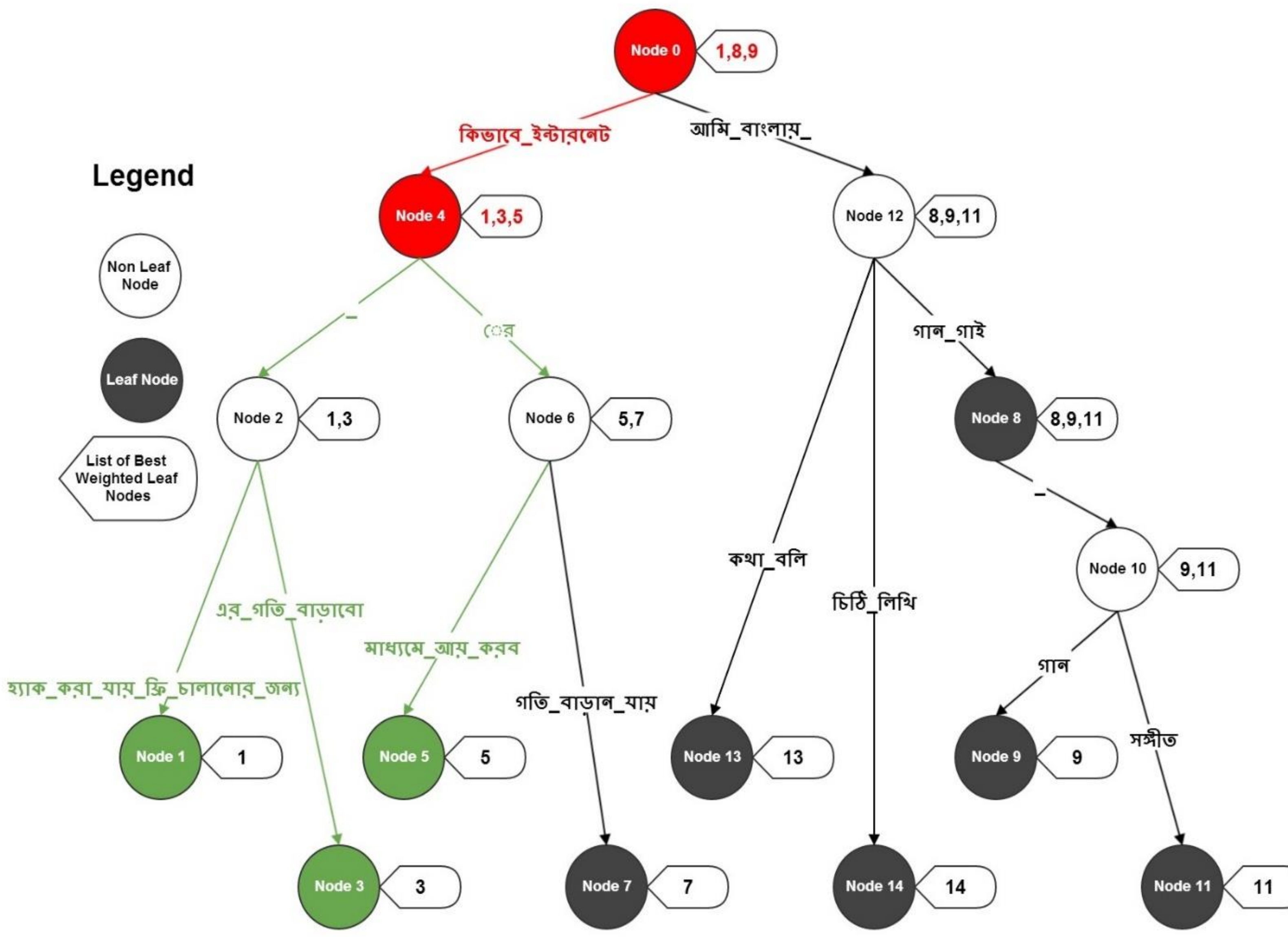
কিভাবে ইন্টারনেট

Suggestion

কিভাবে ইন্টারনেট হ্যাক করা যায় ফ্রি চালানোর জন্য
কিভাবে ইন্টারনেট এর গতি বাড়াবো
কিভাবে ইন্টারনেটের মাধ্যমে আয় করব
কিভাবে ইন্টারনেটের গতি বাড়ান যায়

How this is information is fetched is described below.

- In our compressed trie we try to look up the string “কিভাবে ইন্টারনেট” as close as possible.
- Whenever we find an unmatched character we stop right there.
- We know the path from the root to the last accessible node.
- All the information is fetched from the nodes starting from the root and ending at the last node.
- The last node knows the best possible leaf nodes that we could’ve gone to.So this is given the highest priority among all the other nodes retrieved.
- Thus we get a list of leaf nodes along their weight.
- These leaf nodes are sorted according to their weight calculated.
- Among the candidate leaf nodes we pick the best ones according to their weight.
- These leaf nodes are then traversed right back to the root to get the original string.



The Above figure simulates what we have stated.The simulation procedure is stated below

1. Start from root node(Node 0)
2. Last accessible node is Node 4.
3. Pick the leaf nodes from Node 0 and 4.
4. Node 4 will have higher priority from Node 0 as it is closer to the bottom of the tree.
5. Collect the leaf nodes .They are Node 1,3,5,8,9.
6. But Leaf node 8,9 came from Node 0 thus having less priority.
7. So we select Leaf nodes 1,3,5 sorted according to their weight.
8. Now For each leaf node 1,3,5 traverse backwards right to the top of the tree that is Node 0.
9. We get the result String.

5.5 Complexity Analysis and Memory Usage

5.5.1 Performance

For Each query string checked in Compressed Trie we just need to scan the input string , the results are picked in constant complexity.Thus information is retrieved in O(1) complexity which is the best possible run time for any algorithm.Though there are some constant factors added.For Even further efficiency readers' can minimize the constant factor to get even faster results.

5.5.2 Memory Usage

For each unique query string inserted in Compressed Trie we may have at most 2 nodes (worst case).Thus if there are n unique strings worst case memory usage is 2*n which can stated O(n) in Big-O notation.

5.6 Future Work

There are some possible enhancements that can be done on the above algorithm to get even better accuracy and better run time. Some features can also be included to improve user's experience.They are

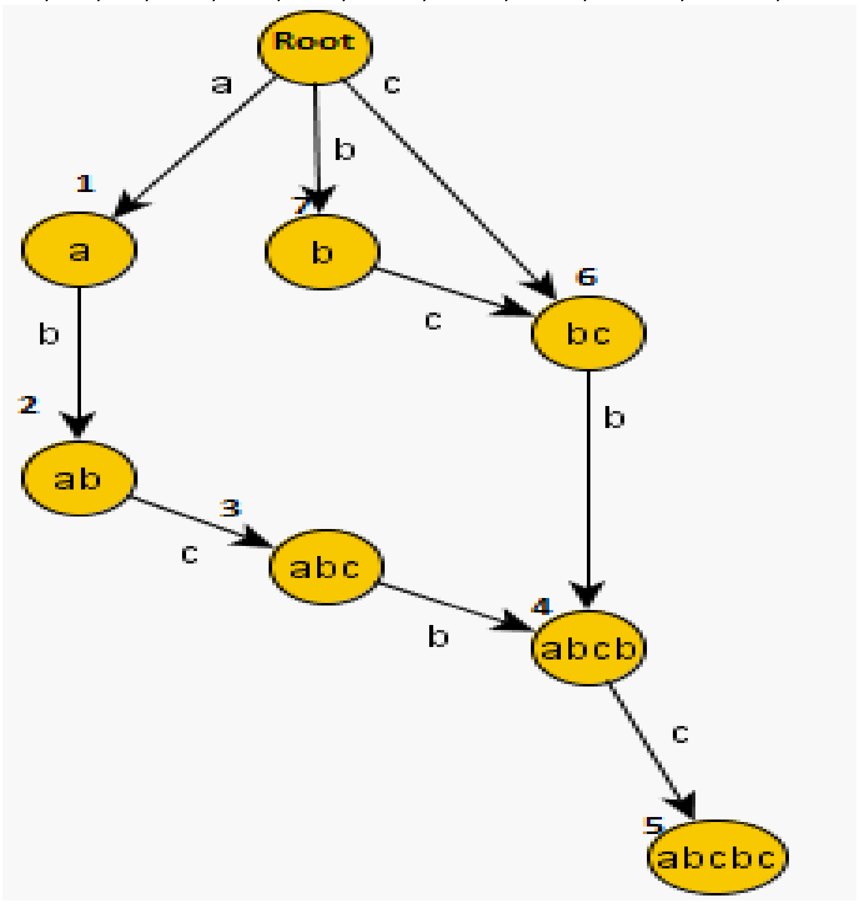
- Query can be personalized for user to user to get improved accuracy
- The Compressed Trie Tree can be stored in distributed servers, thus reducing overhead and storing terabytes of data.
- Zero query search suggestion can be offered to users get better user's experience.

6 Document Based Query Suggestion and Auto Completion

There are millions gigabytes of data over internet. The user normally want to search something which is present in a document. So, it is very much necessary to include even the data of different documents for query suggestion.
A user normally wants the exact matching of the query he types. More precisely he wants to get a suggestion whose prefix is the query he typed. But the number of different set of consecutive words/n-gram in a document is huge. Generally a sentence of n words has close to $n^2/2$ different n-grams in the worst case. So, keeping the information all the n-grams is challenging. So, we need an n-gram model which is memory and time efficient. This n-gram model should also ensure the speed of the runtime search for a query.

6.1 Suffix Automata

Suffix automata is a recent, very popular and widely used data structure and algorithm. It store all the substrings of a string. The time and space complexity is O(n). Simply it creates a DAG (Directed acyclic graph) from a string. For every different way of traversing the graph, we get a different substring.
For example consider a substring “abcbc”. The substrings of this string are, “a”,”b”,”c”,”ab”,”bc”,”cb”,”abc”,”bcb”,”cbc”,”abcb”,”bcbc”,”abcbc”.



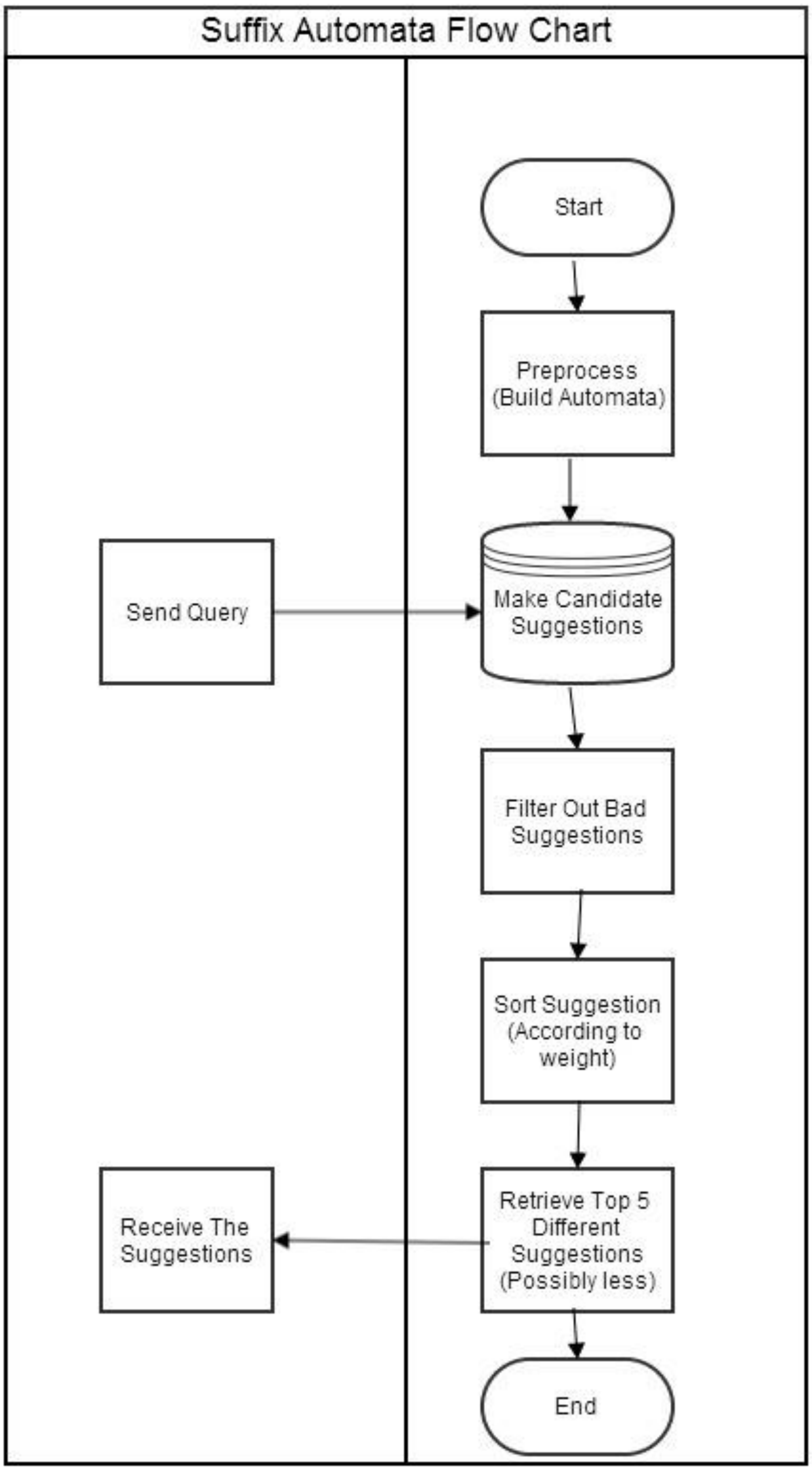
Above Figure makes a DAG of the string using Suffix automata. Here every edge corresponds to an character. We can find any substring of the string by traversing from the root. For example
Path to substring “abc” is (root,1,2,3).
Path to substring “bcb” is (root,7,6,4).
Path to substring ”c” is (root,6).
The automata also keeps the frequency of every substring. So, we can also know the occurrence of any substring. Moreover, we can also apply automata to multiple strings not to mention with the same complexity and considerably less constant factor.

6.2 Why this Algorithm?

Every words can be uniquely identified as an integer. Thus, if we convert the words of a document into integers we get several arrays of integers. Now, as we can make an automata of characters, we can also do the same for integers. Simply, we consider the arrays as several strings whose characters are integers. Moreover, we can easily find the frequency of an n-gram quite easily in runtime. So, from this we can decide which query to show as suggestion. As mentioned before, It also saves a lot of time and memory. The performance comparing time and memory is shown in the performance analysis section.

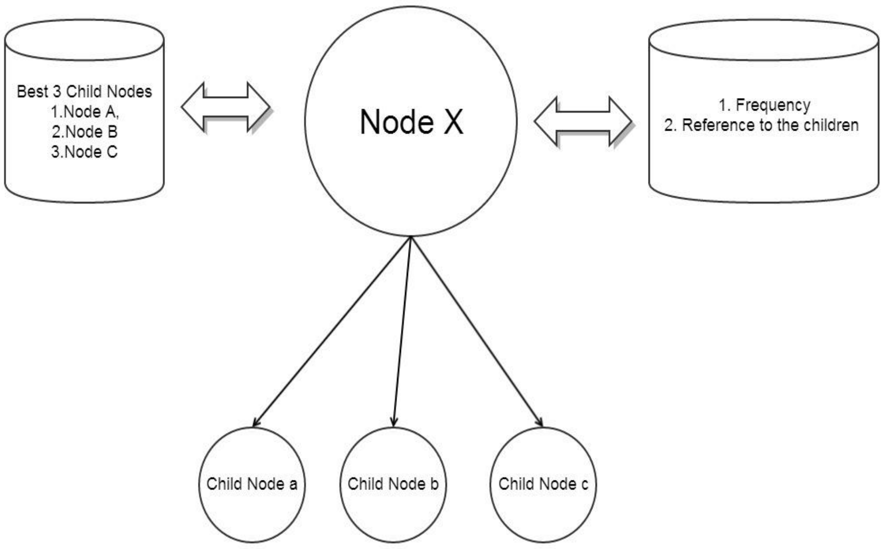
6.3 Description

Below is the flow chart of the system –


Suffix Automata Flow Chart
Start
Preprocess (Build Automata)
Send Query
Make Candidate Suggestions
Filter Out Bad Suggestions
Sort Suggestion (According to weight)
Receive The Suggestions
Retrieve Top 5 Different Suggestions (Possibly less)
End

6.3.1 Preprocess and storage

We mapped every word to a unique integer as mentioned before. Then we build automata on different documents. There is also one extra feature included in this data structure. We keep track of the best 3 (based on frequency) child nodes for every parent node. This serves the purpose of the n-gram of words and helps us to take necessary decision of which way to traverse from a node. One thing to remember is that here each node refers to several (sometimes ever more than 1) queries.



6.3.2 Making candidate suggestions and filtering out bad suggestion

The suggestions for an input query (Q) includes all the candidate query (CQ) whose prefix is Q. For that, we first traverse the automata using the exact query. When we reached the node we take the best 3 child nodes of the node and the best 3 child nodes of the child nodes. Let us name this method Retrieval of direct suggestion from a Node method (RDSN). So, the first candidate of RDSN is the query itself. Moreover, we will also include some extra words in an RDSN candidate following the formula:-

Number of extra words in an RDSN candidate = (number of words in this RDSN candidate-1)/3+1.

Here first word cannot be an extra word. The candidate for extra words are picked from the best 3 child nodes.

Number of words in an RDSN candidate is= (number of words in Q-1) to (number of words in Q+3);

After getting all the candidate suggestions, we filtered out bad suggestions. To filter them out we used 2 criteria. If any of these 2 criteria fails, the suggestion is cancelled. These are

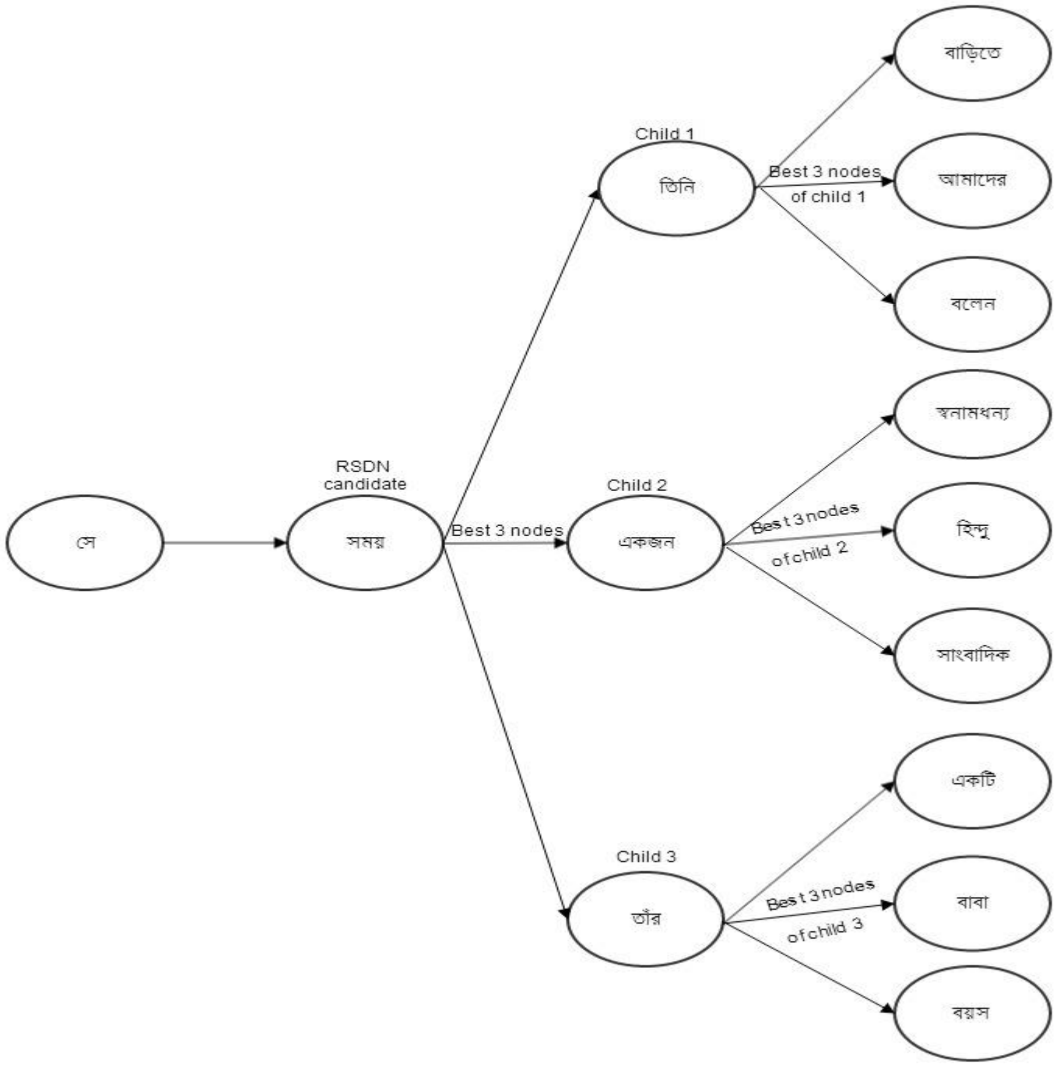
Close Match Ratio, CMR= (number of words both in CQ and Q /number of words in Q).

CMR should be greater than a certain limit.

Weight Match Ratio, WMR= (weight of CQ/ weight of Q). WMR should also be greater than a limit.

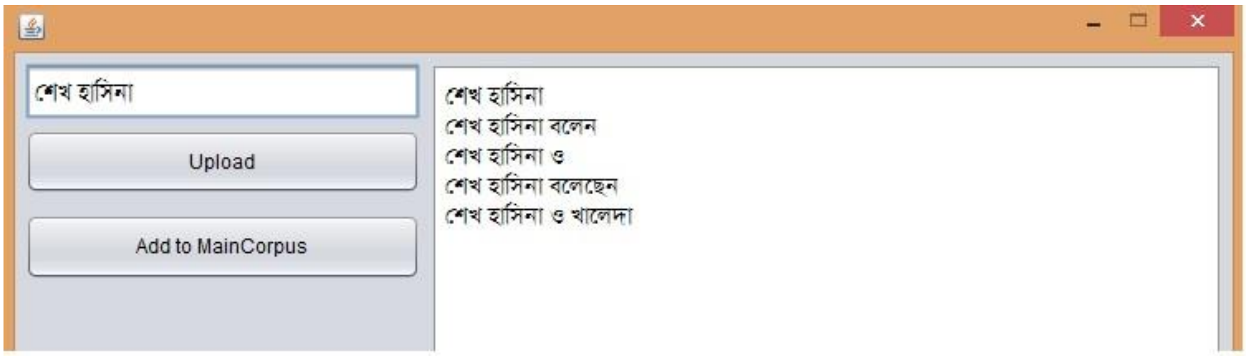
Here Weight of a query, WQ= Sum of the weight of each individual words of the query.

The weight of a word WW which is present in input query**, Q = 1/ the total frequency of the word in all the documents.**
If a word is not present in Q, then WW of the word is 0.



6.3.3 Sorting suggestions

After getting all the valid suggestions, we sort all the suggestion according to some criteria. Every suggestion is given a weighted frequency (WF). Then we sort them in decreasing order of WF value. WF is given following some rules.
Exact Prefix match: If Q is prefix of CQ, it is given the highest weight (by multiplying it with a large number). More precisely the WF of CQ in this case is
Frequency of CQ* Exact match weight.
Here exact match weight is a large constant value.

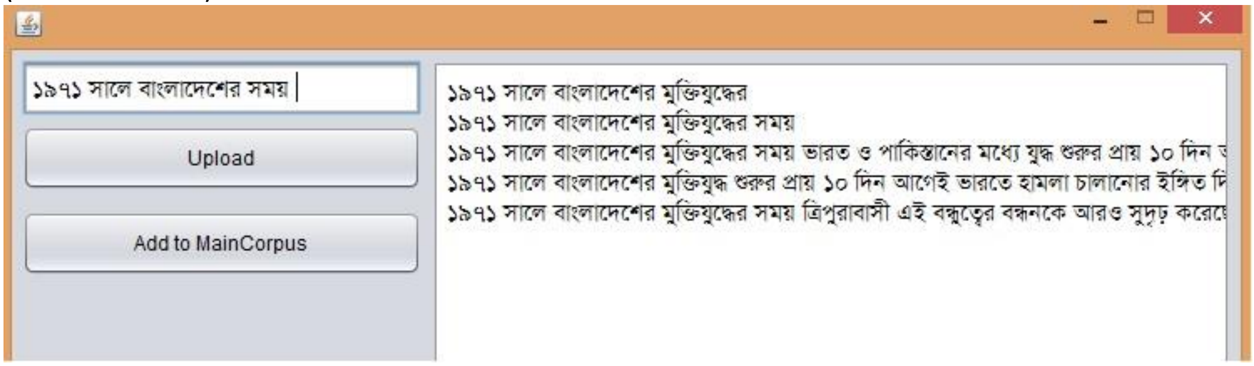


শেখ হাসিনা

শেখ হাসিনা
শেখ হাসিনা বলেন
শেখ হাসিনা ও
শেখ হাসিনা বলেছেন
শেখ হাসিনা ও খালেদা

Partial Prefix match: If Q is partially prefix of CQ then WF of CQ is the frequency of CQ* Partial match weight (constant value). A CQ meets this criteria if WMR of the common prefix of Q and CQ is greater than a certain limit.

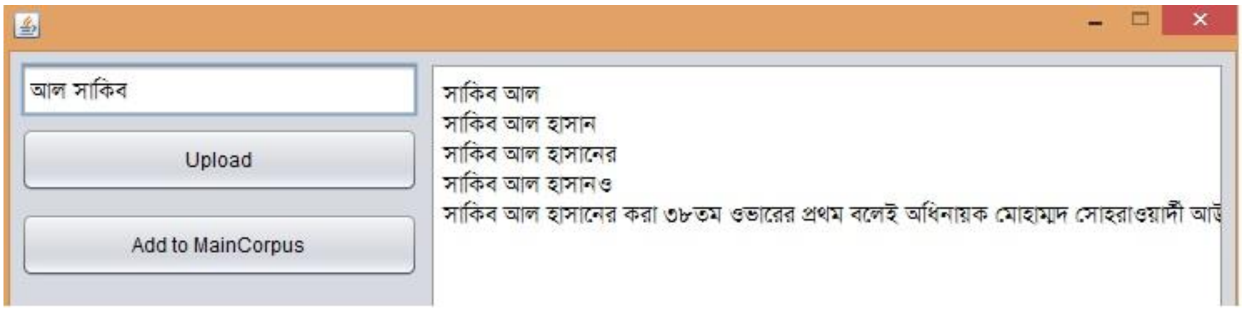
Serial Match: If Q is not a partial prefix, then we check if Q is closely a subsequence of CQ. Let us define Serial Match Value, SMV = number of words in Q which serially matches with CQ. If SMV is greater than a certain value then CQ is considered a serial match.
So, in this case WF of CQ = frequency of CQ*Serial match value* Serial match weight (constant value).



১৯৭১ সালে বাংলাদেশের সময়

১৯৭১ সালে বাংলাদেশের মুক্তিযুদ্ধের
১৯৭১ সালে বাংলাদেশের মুক্তিযুদ্ধের সময়
১৯৭১ সালে বাংলাদেশের মুক্তিযুদ্ধের সময় ভারত ও পাকিস্তানের মধ্যে যুদ্ধ শুরুর প্রায় ১০ দিন
১৯৭১ সালে বাংলাদেশের মুক্তিযুদ্ধ শুরুর প্রায় ১০ দিন আগেই ভারতে হামলা চালানোর ইঙ্গিত
১৯৭১ সালে বাংলাদেশের মুক্তিযুদ্ধের সময় ত্রিপুরাবাসী এই বন্ধুত্বের বন্ধনকে আরও সুদৃঢ় করে

Good WMR and CMR values of a CQ: It may happen that a CQ fails all the 3 above criteria but it still may be a good suggestion for CQ. For example, a user can make a query for "হাসিনা শেখ", a good suggestion for the query is "শেখ হাসিনা". But this suggestion fails above 3 criteria and only in the last case it passes. So, this kind of suggestion is very much crucial even for grammar check (mentioned later).

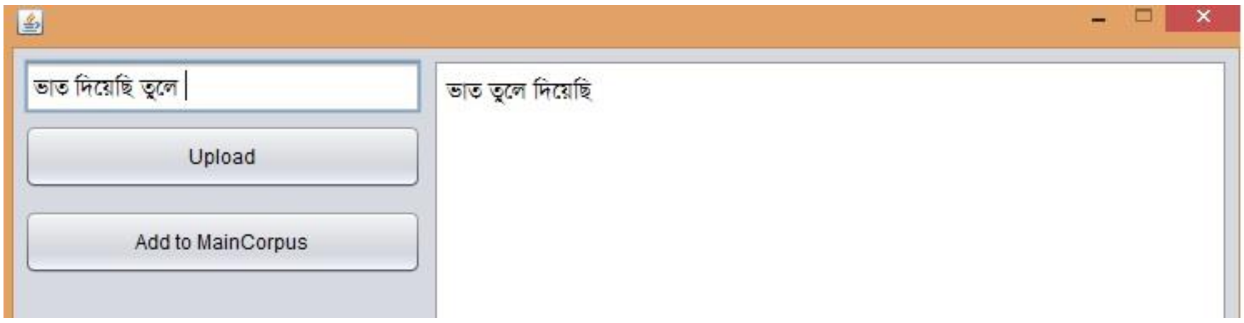


আল সাকিব

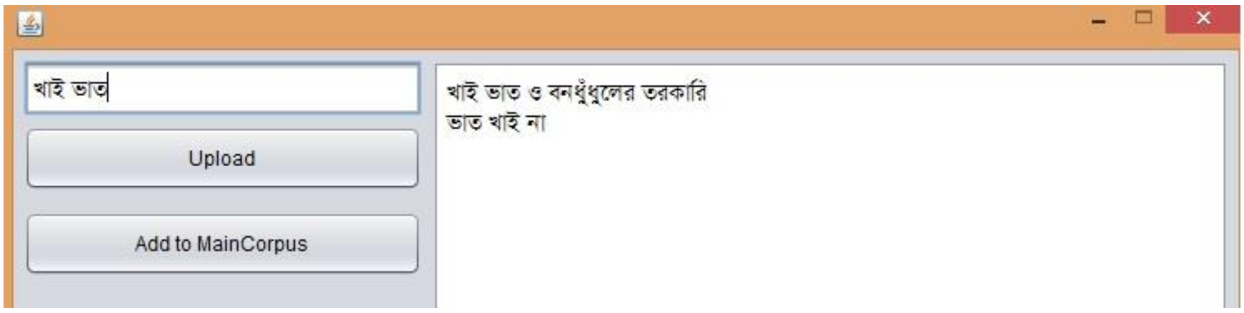
সাকিব আল
সাকিব আল হাসান
সাকিব আল হাসানের
সাকিব আল হাসানও
সাকিব আল হাসানের করা ৩৮তম ওভারের প্রথম বলেই অধিনায়ক মোহাম্মদ সোহরাওয়ার্দী

After getting all the sorted suggestions, one other thing we keep in mind and that is, there can be several CQs which have same WF but are the prefix of another CQ. That only means that the query come from the same source. So, we should cancel out the prefix matching ones.
After all the above tasks, we just send the top 5 CQs to the main system.

Grammar check: There is one extra feature which is grammatical error check. If there is enough data often the system also suggests a user to use grammatically correct suggestions.


ভাত দিয়েছি তুলে

ভাত তুলে দিয়েছি


খাই ভাত

খাই ভাত ও বনধুঁধুলের তরকারি
ভাত খাই না

6.4 Performance Analysis

Data Size	Memory Usage	Time	Memory/MB	Time/MB
35 MB	1491 MB	3344ms	42.6 MB	95ms
66 MB	2322 MB	6041ms	35.18 MB	91ms
112 MB	3034 MB	9344ms	27.09 MB	83ms
175 MB	4063 MB	12613ms	23.22 MB	72ms

Though asymptotically the time and memory complexity of preprocessing automata is O (n). The graph is not linear. In fact the memory and time needed for per megabyte of data decreases as data set increases. So, we can use huge data for preprocessing purpose. So, this algorithm always works better for bigger corpus. Moreover, the runtime complexity can be viewed as 0(1) as the complexity is equal to the number of words in the input query. These are tested on a Core-I5 pc with 8GB RAM.

6.5 Future Work

There are some modifications which can be applied in future versions. First of all, we can make suggestion using the root word. For example “আমার” “আমাদের”, “আমি” may mean the same word. Secondly, we may also increase the weight of the n-grams of the most popular and the recently visited documents.

7. Final Result

We have stated how we have fetched relevant Search Suggestion and Auto Completion using Suffix Automata and Compressed Trie algorithm. But the results of this two approaches is significantly different. Because

1. Compressed Trie is built using Query Log Data.
2. Suffix Automata is built using Document Data.
3. The size and quality of the source of data from different source is quite different.
4. The data are themselves very much different.
5. Query Log is the actual query data which users search.
6. Document data is the data where massive information is found.

Thus some measures must be taken in order to merge both the results of this two algorithms and display top most relevant results.

7.1 Solution

Both Compressed Trie and Suffix automata has their unique Weight Calculation method but the fundamental scoring system of both of them is same. But size of the corpus data that is being fed to these machine is different. Thus for each search suggestion and auto completion provided by these two algorithms has to go undergo another step which is.

1. For Each Auto Completion provided by the Compressed Trie, its weight $w1$ has to be divided by $\log_2(total\ number\ of\ nodes\ in\ compressed\ trie)$ i.e the $actual\ weight\ x1 = \frac{w1}{\log_2(N1)}$

2. For Each Suggestion provided by the Suffix Automata, its weight $w2$has to be divided by $\log_2(total\ number\ of\ nodes\ in\ suffix\ automata)$ i.e $actual\ weight\ x2 = \frac{w2}{\log_2(N2)}$

8. Conclusion

This thesis is a good experience for us. Suggesting a query for a user is a challenging task. Hope this proves to be a user friendly system and thus make Bangla search engine related experience more fruitful in the coming days.

9. References

1. http://t2a.co/blog/index.php/spell-checking-or-search-engine-suggestions-using-bk-trees/
2. http://hamberg.no/erlend/posts/2012−01−17−BK-trees.html
3. http://blog.notdot.net/2007/4/Damn-Cool-Algorithms-Part-1-BK-Trees
4. http://en.wikipedia.org/wiki/N-gram
5. http://www.informationisbeautiful.net/visualizations/google-ngram-experiments/
6. https://books.google.com/ngrams
7. http://www.mathcs.emory.edu/~cheung/Courses/323/Syllabus/Text/trie02.html
8. http://e-maxx.ru/algo/suffix_automata