C++ PROGRAM

OF

WORD PREDICTION APPLICATION

A PROJECT
REPORT BY

UTKARSH TYAGI

(18BCE0758)

KARTIKAY GUPTA

(18BCE2199)

Submitted
to:Prof.Govinda K

DATA STRUCTURES AND ALGORITHMS (CSE2003)

March 2019


VELLORE INSTITUTE OF TECHNOLOGY
VIT
VIT®
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

March 2019

AIM:

To understand the dynamic data structure TRIE based on it develop a program having industrial application to predict words .

OBJECTIVES:

- To understand the dynamic data structure tree used in developing the program
- To understand the data structure ‗trie' being used in the program.
- To construct a strong and efficient algorithm to develop the program which is editable and can be later used as a module for bigger software mechanism
- To develop a real time program which is efficient and has a fast processing and also has a industrial application.

ABSTRACT:

In this busy world no one has time now. Technology is being developed every day to increase the efficiency. In this front, word predictor is a small step which increases our efficiency multifold times.

Word predictor has applications in various areas like texting, search engine etc. To develop our word predictor program they have used the data structure Trie and second approach is n grams algorithms. Our program uses a stored file of words to predict the words which the user may think of thus helping a lot.

INTRODUCTION:

Autocomplete, or word completion, is a feature in which an application predicts the rest of a word a user is typing. In graphical user interfaces, users can typically press the tab key to accept a suggestion or the down arrow key to accept one of several.

Autocomplete speeds up human-computer interactions when it correctly predicts the word a user intends to enter after only a few characters have been typed into a text input field. It works best in domains with a limited number of possible words (such as in command line interpreters), when some words are much more common (such as when addressing an email), or writing structured and predictable text (as in source code editors).

Many autocomplete algorithms learn new words after the user has written them a few times, and can suggest alternatives based on the learned habits of the individual user.

Autocomplete or word completion works so that when the writer writes the first letter or letters of a word, the program predicts one or more possible words as choices. If the word he intends to –write is included in the list he can select it, for example by using the number keys.[1]

If the word that the user wants is not predicted, the writer must enter the next letter of the word. At this time, the word choice(s) is altered so that the words provided begin with the same letters as those that have been selected. When the word that the user wants appears it is selected, and the word is inserted into the text.

In another form of word prediction, words most likely to follow the just written one are predicted, based on recent word pairs used. Word prediction uses language modeling, where within a set vocabulary the words are most likely to occur are calculated. Along with language modeling, basic word prediction on AAC devices is often coupled with a recency model, where

words that are used more frequently by the AAC user are more likely to be predicted. Word prediction software often also allows the user to enter their own words into the word prediction dictionaries either directly, or by "learning" words that have been written.[2]

DATA STRUCTURE TRIE

In this program Data structure Trie is being used to search the data in an ordered fashion. In computer science, a trie, also called digital tree and sometimes radix tree or prefix tree (as they can be searched by prefixes), is a kind of search tree—an ordered tree data structure that is used to store a dynamic set or associative array where the keys are usually strings.[3]

Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree defines the key with which it is associated. All the descendants of a node have a common prefix of the string associated with that node, and the root is associated with the empty string.
Values are not necessarily associated with every node. Rather, values tend only to be associated with leaves, and with some inner nodes that correspond to keys of interest.

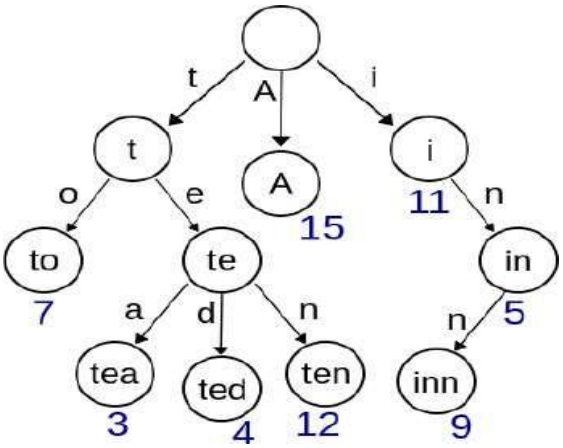


Figure1.
For the space-optimized presentation of prefix tree, see compact prefix tree. In the example shown, keys are listed in the nodes and values below them.[4]
Each complete English word has an arbitrary integer value associated with it. A trie can be seen as a tree-shaped deterministic finite automaton. Each finite language is generated by a trie automaton, and each trie can be compressed into a deterministic acyclic finite state automaton.[5]

LITERATURE REVIEW:

In computer science, binary search, also known as half-interval search, logarithmic search, or binary chop, is a search algorithm that finds the position of a target value within a sorted array. Binary search compares the target value to the middle element of the array. If they are not equal, the half in which the target cannot lie is eliminated and the search continues on the remaining half, again taking the middle element to compare to the target value, and repeating this until the target value is found. If the search ends with the remaining half being empty, the target is not in the array. Even though the idea is simple, implementing binary search correctly requires attention to some subtleties about its exit conditions and midpoint calculation, particularly if the values in the array are not all of the whole numbers in the range.

Binary search runs in logarithmic time in the worst case, making O(log n) comparisons, where n is the number of elements in the array, the O is Big O notation, and log is the logarithm. Binary search takes constant (O(1)) space, meaning that the space taken by the algorithm is the same for any number of elements in the array. Binary search is faster than linear search except for small arrays, but the array must be sorted first. Although specialized data structures designed for fast searching, such as hash tables, can be searched more efficiently, binary search applies to a wider range of problems.[6]

There are numerous variations of binary search. In particular, fractional cascading speeds up binary searches for the same value in multiple arrays. Fractional cascading efficiently solves a number of search problems in computational geometry and in numerous other fields. Exponential search extends binary search to unbounded lists. The binary search tree and B-tree data structures are based on binary search.

Writing is a multifaceted, complex task that involves interaction between physical and cognitive skills. Individuals with physical disabilities vary in terms of both their physical and cognitive abilities. Often they must overcome one or more significant barriers in order to engage in the task of writing.[7]

Minimizing or eliminating barriers is important because opportunities are greater for individuals who can effectively communicate their ideas via writing. Assistive technology (AT) is an increasingly effective solution to increase typing fluency. The purpose of this study is to examine if word prediction software, a commonly used software program used with individuals with learning disabilities, will be

effective for those with physical impairments to increase typing rate and reduce spelling errors (fluency).

Data will be collected for words correct per minute (WCPM) and errors (e.g., spelling). Four middle- or high school-aged participants with diverse physical disabilities will be recruited in this single subject, alternating treatment design.

Participants will type for three-minute timed sessions using either a standard word processor or Co:Writer 4000, a word prediction software program.[8]
Specific research questions are: (a) to what extent will students with physical and health disabilities produce greater WCPM when writing a draft paper on a common topic using word prediction rather than word processing.
As the library website and its online searching tools become the primary —branch|many users visit for their research, methods for providing automated, context-sensitive research assistance need to be developed to guide unmediated searching toward the most relevant results. This study examines one such method, the use of autocompletion in search interfaces, by conducting usability tests on its use in typical academic research scenarios.
The study reports notable findings on user preference for autocomplete features and suggests best practices for their implementation.
We consider the following autocompletion search scenario: imagine a user of a search engine typing a query; then with every keystroke display those completions of the last query word that would lead to the best hits, and also display the best such hits.

The following problem is at the core of this feature: for a fixed document collection, given a set D of documents, and an alphabetical range W of words, compute the set of all word-in-document pairs (w, d) from the collection such that $w \in W$ and $d \in D$. We present a new data structure with the help of which such autocompletion queries can be processed, on the average, in time linear in the input plus output size, independent of the size of the underlying document collection.[9]

At the same time, our data structure uses no more space than an inverted index. Actual query processing times on a large test collection correlate almost perfectly with our theoretical bound.

Predictive text completion is a technology that extends the traditional autocompletion and text replacement techniques. It helps to reduce key strokes needed in text input and serves as a more affordable assistive technology for computer users who are

dyslexic or has learning/reading difficulty/disability, as compared to more expensive speech-to-text technology or special input devices. Python is used for prototyping, rapid R&D and testing of advanced features, while AutoHotKey scripting language can be used to code the regular stable release.[10]

PROPOSED WORK:

PSEUDO CODE:

Start

Create a class named Node which have mcontent to store an alphabet, mmarker a Boolean variablie used to check wheather a word ends there , mChildren as a vector of nodes.

Create a constructor for class to initialize mcontent as blank space and mMarker as false.

Define a member function named content which returns the mcontent which is the alphabet stored in that node.

Define a member function named setContent which assigns the alphabet to mContent of that node.

Define a member function of class node named wordMarker which returns mMarker i.e.,it return true if any word ends at that alphabet else returns false.

Define a member function of class node named setWordMarker which assigns true to mMarker of that node.

Define a member function named appendChild which adds another node to is children vector mChildren.

Define a member function named findChild, in this function it stores each node of its children vector in temp node and check wheather the mContent of temp node matches with alphabet that is passed through function findChild, if it matches then the temp node is returned.

If it does not match with any mContent of its children nodes then NULL is returned.

Create another class named Trie which have variable named root which is an object of node class.

Define a function named addWord of class Trie it takes a string s while calling the function. Then root node is stored in current.

Then it takes the first letter of string s i.e, s[0], searches s[0] in children of currnent with help of findChild(s[0]) function and stores the return node in child.

If the child is NULL then it means s[0] is not in the children of current else child contains a node whose mContent is equal to s[0].

If child is not Null then child node is assigned to current

else if child is NULL then a temp node is created and s[0] is assigned to mContent with setContent function and tmp node is added to the children

vector of current node with help of appendChild function and then temp node is assigned to current.

Then this process is continued for all the letters of string so the word is added to tree if it is not there.

Define another function named autoComplete of class Trie which takes a string as s and a vector of strings as res. Then store root node in current node.

Then it takes the first letter of string s i.e, s[0], searches s[0] in children of currnent with help of findChild(s[0]) function and stores the return node in tmp.

If the temp is NULL then it means s[0] is not in the children of current else tmp contains a node whose mContent is equal to s[0].

If tmp is NULL then return NULL

else store temp node to current

Repeat the same process for all letters until end of word. Then

copy the string s into another string c.

Then declare a variable named loop and set it to true.

Then execute parseTree function in it.

Define another function named parseTree with parameters current as node, s as string, res as vector of strings and loop as bool.

Declare two character array variables as k and a with arrays filled with 0

If loop is true , current is not null and mMarker of current node is true which means a word is ending there then add the string s in res and then check the size of res. If the size exceeds some particular number like 15 then set loop to false.

Create a vector of nodes named child and store children vector of current in it.

Then run a a for loop for size of child times. In the loop copy strings to string k.

then store child[i] mContent in a and then concatenate in k and a

Then if loop is true then again do function parseTree by passing child[i] as node and k as string

So this function stores all possible words in res vector.

Define another function named loadDictionary which takes all the word in text file and add them to a tree starting with rood node.

In main function first load the dictionary using loadDictionary function. Then a menu is displayed with two options

1) Auto complete

2)Quit

Then read mode as option number.

By using switch case do the selsected case. In autocomplete case read a strings convert them into lower

Then create an vector of strings named auto Complete List.

Then execute autocomplete function by passing s and auto Complete List in it. Then if auto Complete List is empty print No suggestions

Else print all the word in auto Complete List In

Exit case just return 0 to exit main function End

RESULTS AND DISCUSSION:

```
#include<iostream>
#include<fstream>
#include<vector>
#include<string>
#include<set>
#include<algorithm>
#include<string.h>
#include<iomanip> using
namespace std; class
Node
{
public:
Node()
{
mContent = ' ';
mMarker = false;.
}
~Node() {} char
content()
{
return mContent;
}
void setContent(char c)
{
mContent = c;
}
bool wordMarker()
{
return mMarker;
}
void setWordMarker()
{
mMarker = true;
}
Node* findChild(char c);
void appendChild(Node* child)
```

```
{
mChildren.push_back(child);
}
vector<Node*> children()
{
return mChildren;
}
private:
char mContent;
bool mMarker;
vector<Node*> mChildren;
};
Node* Node::findChild(char c)
{
for ( int i = 0; i < mChildren.size(); i++ )
{
Node* tmp = mChildren.at(i); if (
tmp->content() == c )
{
return tmp;
}
}
return NULL;
}
class Trie
{
public: Trie();
~Trie();
void addWord(string s);
bool searchWord(string s);
bool autoComplete(string s,vector<string>&);
void parseTree(Node *current,char * s,vector<string>&,bool &loop); private:
Node* root;
};
Trie::Trie()
{
root = new Node();
}
Trie::~Trie()
{
// Free memory
}
void Trie::addWord(string s)
{
Node* current = root; if(
s.length()==0)
{
current->setWordMarker();
return;
}
for ( int i = 0; i < s.length(); i++ )
{
Node*child=current->findChild(s[i]); if (
child != NULL )
```

```
{
current = child;
}
else
{
Node* tmp = new Node();
tmp->setContent(s[i]); current-
>appendChild(tmp); current =
tmp;
}
if ( i == s.length() - 1 )
current->setWordMarker();
}
}
bool Trie::searchWord(string s)
{
Node* current = root;
while(current != NULL)
{
for ( int i = 0; i < s.length(); i++ )
{
Node* tmp = current->findChild(s[i]); if (
tmp == NULL )
return false;
current = tmp;
}
if(current->wordMarker())
return true;
else
returnfalse;
}
returnfalse;
}
bool Trie::autoComplete(std::string s, std::vector<string> &res)
{
Node *current=root;
for ( int i = 0; i < s.length(); i++ )
{
Node* tmp = current->findChild(s[i]); if (
tmp == NULL )
return false;
current = tmp;
}
char c[100];
strcpy(c,s.c_str());
bool loop=true;
parseTree(current,c,res,loop);
return true;
}
void Trie::parseTree(Node *current, char *s,std::vector<string> &res,bool& loop)
{
char k[100]= {0};
char a[2]= {0};
if(loop)
{
if(current!=NULL)
```

```
{
if(current->wordMarker()==true)
{
res.push_back(s);
if(res.size()>15)
loop=false;
}
vector<Node *> child=current->children(); for(int
i=0; i<child.size() && loop; i++)
{
strcpy(k,s); a[0]=child[i]-
>content(); a[1]='\0';
strcat(k,a);
if(loop)
parseTree(child[i],k,res,loop);
}
}
}
}
bool loadDictionary(Trie* trie,string filename)
{
ifstream words; ifstream
input;
words.open(filename.c_str());
if(!words.is_open())
{
cout<<"Dictionary file Not Open"<<endl; return
false;
}
while(!words.eof())
{
char s[100];
words >> s;
trie->addWord(s);
}
return true;
}
int main()
{
system("color 1E");
Trie* trie = new Trie(); int
mode;
cout<<"Loading dictionary"<<endl;
loadDictionary(trie,"wordlist.txt"); while(1)
{
cout<<endl<<endl;
cout<<"Interactive mode,press "<<endl;
cout<<"1: Auto Complete Feature"<<endl;
cout<<"2: Quit"<<endl<<endl; cin>>mode;
switch(mode)
{
case 1://Auto complete
{
```

```
string s; cin>>s;
transform(s.begin(), s.end(), s.begin(), ::tolower);
vector<string> autoCompleteList;
trie->autoComplete(s,autoCompleteList);
if(autoCompleteList.size()==0)
{
cout<<"No suggestions"<<endl;
}
else
{
cout<<"Autocomplete reply :"<<endl; for(int i=0;
i<autoCompleteList.size(); i++)
{
cout<<"\t \t "<<autoCompleteList[i]<<endl;
}
}
}
continue;
case 2:
delete trie;
return 0;
default:
continue;
}
}
}
```

OUTPUT:

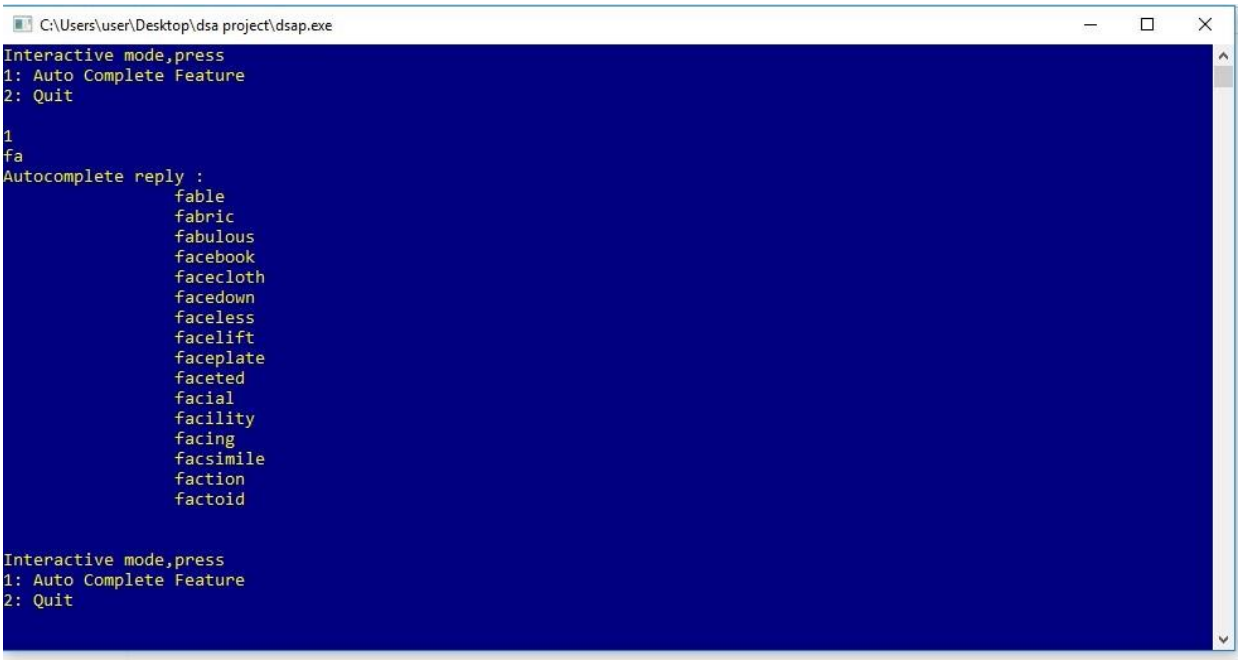


Figure2.

Figure3.

In the above output the program is being implemented to give the words predicted. First the program confirms the loading of dictionary file.
Interactive mode allows us to use word predictor or quit the program When chose to predict word PLA
Some 16 results are predicted with respect to PLA
Simailarly when predict word APP
Again 16 results are predicted with respect to APP
The extra text file containing the lit of words can be downloaded from the internet.

GRAPHS AND ANALYSIS

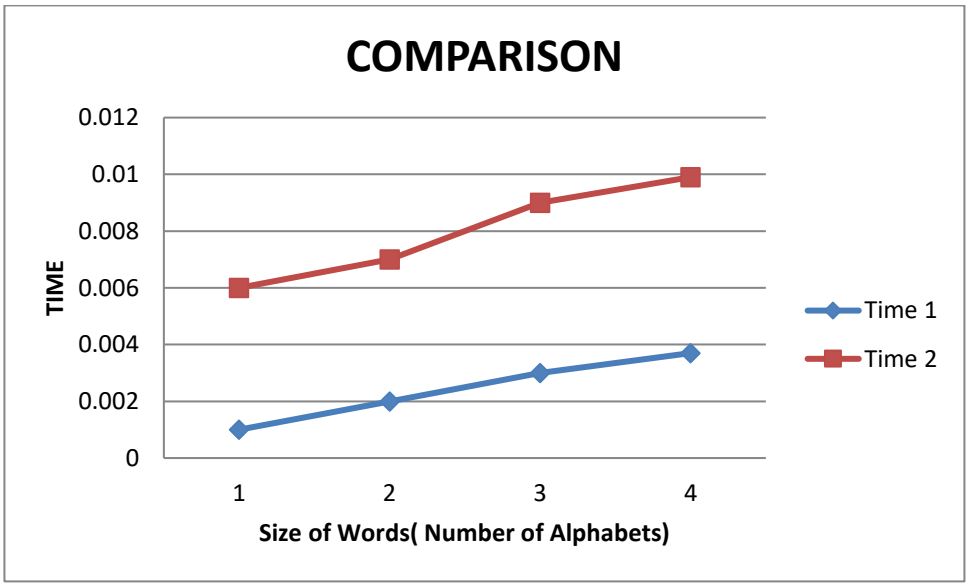


Figure4.

APPROACH 2(N-grams)

Literature Review

In the fields of computational linguistics and probability, an n-gram is a contiguous sequence of n items from a given sample 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. When the items are words, n-grams may also be called shingles.[11]

Using Latin numerical prefixes, an n-gram of size 1 is referred to as a "unigram"; size 2 is a "bigram" (or, less commonly, a "digram"); size 3 is a "trigram". English cardinal numbers are sometimes used, e.g., "four-gram", "five-gram", and so on. In computational biology, a polymer or oligomer of a known size is called a k-mer instead of an n-gram, with specific names using Greek numerical prefixes such as "monomer", "dimer", "trimer", "tetramer", "pentamer", etc., or English cardinal numbers, "one-mer", "two-mer", "three-mer", etc.[12]

An n-gram is a sub-sequence of n items from a given sequence. n-grams are used in various areas of statistical natural language processing and genetic sequence analysis. The items in question can be characters, words or base pairs according to the application. For example, the sequence of characters "Hatem mostafa helmy" has a 3-gram of ("Hat", "ate", "tem", "em ", "m m", ...), and has a 2-gram of ("Ha", "at", "te", "em", "m ", " m", ...). This n-gram output can be used for a variety of R&D subjects, such as Statistical machine translation and Spell checking.[13]

Pattern extraction is the process of parsing a sequence of items to find or extract a certain pattern of items. Pattern length can be fixed, as in the n-gram model, or it can be variable. Variable length patterns can be directives to certain rules, like regular expressions. They can also be random and depend on the context and pattern repetition in the patterns dictionary.[14]

The algorithm introduced here is derived from the LZW compression algorithm, which includes a magic idea about generating dictionary items at compression time while parsing the input sequence. If you have no idea about LZW, you can check it out at my article, Fast LZW compression. And of course, the algorithm inherits the speed of my implementation to LZW, plus extra speed for two reasons:

1. The parsing item is a word, not a letter
2. There's no destination buffer, as there is no need for a compressed buffer.[15]

The algorithm uses a binary tree to keep extracted patterns that give the algorithm excellent speed at run-time to find and fetch new items to the dictionary. Let us discuss the algorithm pseudo code. We have some figures to clarify the idea with an algorithm flow chart and a simple example.[16]

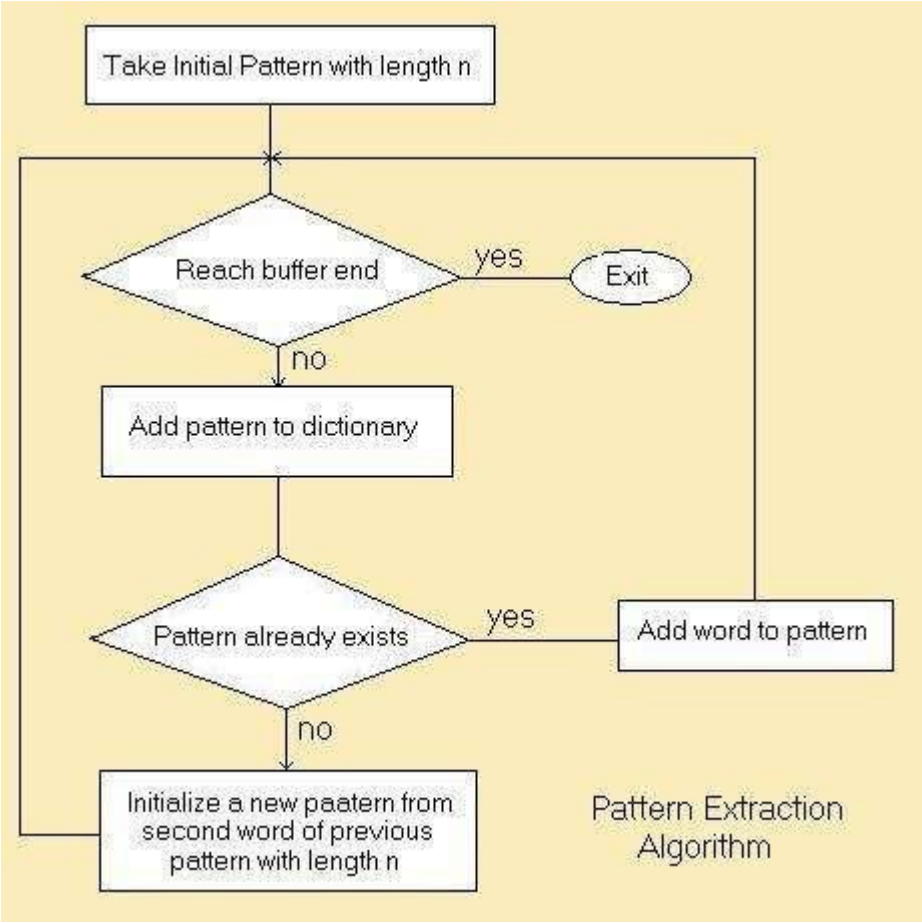
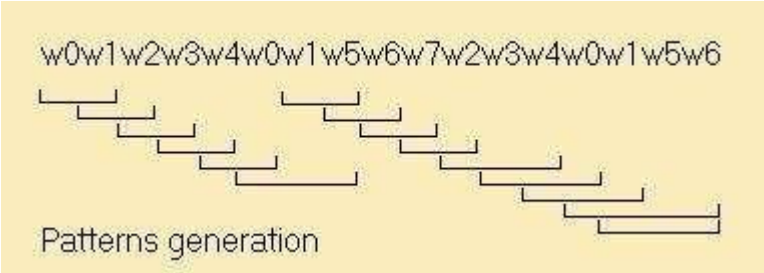


Figure5
Example: the input words sequence w0w1w2w3w4w0w1w5w6w7w2w3w4w0w1w5w6

Assume n equals 2; then the initial pattern will be w0w1. After applying the algorithm steps, the resultant dictionary would be as in the fourth column:



Input Sequence	Pattern	Step	Dictionary	Frequency
w0w1w2w3w4w0w1w5w6w7w2w3w4w0w1w5w6	w0w1	Words available?		
w0w1w2w3w4w0w1w5w6w7w2w3w4w0w1w5w6	w0w1	Add to dictionary	w0w1	3
w0w1w2w3w4w0w1w5w6w7w2w3w4w0w1w5w6	w0w1	Pattern exists?		
w1w2w3w4w0w1w5w6w7w2w3w4w0w1w5w6	w1w2	Take new pattern		
w1w2w3w4w0w1w5w6w7w2w3w4w0w1w5w6	w1w2	Words available?		
w1w2w3w4w0w1w5w6w7w2w3w4w0w1w5w6	w1w2	Add to dictionary	w1w2	1
w1w2w3w4w0w1w5w6w7w2w3w4w0w1w5w6	w1w2	Pattern exists?		
w2w3w4w0w1w5w6w7w2w3w4w0w1w5w6	w2w3	Take new pattern		
w2w3w4w0w1w5w6w7w2w3w4w0w1w5w6	w2w3	Words available?		
w2w3w4w0w1w5w6w7w2w3w4w0w1w5w6	w2w3	Add to dictionary	w2w3	2
w2w3w4w0w1w5w6w7w2w3w4w0w1w5w6	w2w3	Pattern exists?		
w3w4w0w1w5w6w7w2w3w4w0w1w5w6	w3w4	Take new pattern		
w3w4w0w1w5w6w7w2w3w4w0w1w5w6	w3w4	Words available?		
w3w4w0w1w5w6w7w2w3w4w0w1w5w6	w3w4	Add to dictionary	w3w4	2
w3w4w0w1w5w6w7w2w3w4w0w1w5w6	w3w4	Pattern exists?		
w4w0w1w5w6w7w2w3w4w0w1w5w6	w4w0	Take new pattern		
w4w0w1w5w6w7w2w3w4w0w1w5w6	w4w0	Words available?		
w4w0w1w5w6w7w2w3w4w0w1w5w6	w4w0	Add to dictionary	w4w0	2
w4w0w1w5w6w7w2w3w4w0w1w5w6	w4w0	Pattern exists?		
w0w1w5w6w7w2w3w4w0w1w5w6	w0w1	Take new pattern		
w0w1w5w6w7w2w3w4w0w1w5w6	w0w1	Words available?		
w0w1w5w6w7w2w3w4w0w1w5w6	w0w1	Add to dictionary		
w0w1w5w6w7w2w3w4w0w1w5w6	w0w1	Pattern exists?		
w0w1w5w6w7w2w3w4w0w1w5w6	w0w1w5	Add word to pattern		
w0w1w5w6w7w2w3w4w0w1w5w6	w0w1w5	Words available?		
w0w1w5w6w7w2w3w4w0w1w5w6	w0w1w5	Add to dictionary	w0w1w5	2
w0w1w5w6w7w2w3w4w0w1w5w6	w0w1w5	Pattern exists?		
w1w5w6w7w2w3w4w0w1w5w6	w1w5	Take new pattern		
w1w5w6w7w2w3w4w0w1w5w6	w1w5	Words available?		
w1w5w6w7w2w3w4w0w1w5w6	w1w5	Add to dictionary	w1w5	2

Input Sequence	Pattern	Step	Dictionary	Frequency
w1w5w6w7w2w3w4w0w1w5w6	w1w5	Pattern exists?		
w5w6w7w2w3w4w0w1w5w6	w5w6	Take new pattern		
w5w6w7w2w3w4w0w1w5w6	w5w6	Words available?		
w5w6w7w2w3w4w0w1w5w6	w5w6	Add to dictionary	w5w6	1
w5w6w7w2w3w4w0w1w5w6	w5w6	Pattern exists?		
w6w7w2w3w4w0w1w5w6	w6w7	Take new pattern		
w6w7w2w3w4w0w1w5w6	w6w7	Words available?		
w6w7w2w3w4w0w1w5w6	w6w7	Add to dictionary	w6w7	1
w6w7w2w3w4w0w1w5w6	w6w7	Pattern exists?		
w7w2w3w4w0w1w5w6	w7w2	Take new pattern		
w7w2w3w4w0w1w5w6	w7w2	Words available?		
w7w2w3w4w0w1w5w6	w7w2	Add to dictionary	w7w2	1
w7w2w3w4w0w1w5w6	w7w2	Pattern exists?		
w2w3w4w0w1w5w6	w2w3	Take new pattern		
w2w3w4w0w1w5w6	w2w3	Words available?		
w2w3w4w0w1w5w6	w2w3	Add to dictionary		
w2w3w4w0w1w5w6	w2w3	Pattern exists?		
w2w3w4w0w1w5w6	w2w3w4	Add word to pattern		
w2w3w4w0w1w5w6	w2w3w4	Words available?		
w2w3w4w0w1w5w6	w2w3w4	Add to dictionary	w2w3w4	1
w2w3w4w0w1w5w6	w2w3w4	Pattern exists?		
w3w4w0w1w5w6	w3w4	Take new pattern		
w3w4w0w1w5w6	w3w4	Words available?		
w3w4w0w1w5w6	w3w4	Add to dictionary		
w3w4w0w1w5w6	w3w4	Pattern exists?		
w3w4w0w1w5w6	w3w4w0	Add word to pattern		
w3w4w0w1w5w6	w3w4w0	Words available?		
w3w4w0w1w5w6	w3w4w0	Add to dictionary	w3w4w0	1
w3w4w0w1w5w6	w3w4w0	Pattern exists?		
w4w0w1w5w6	w4w0	Take new pattern		

Input Sequence	Pattern	Step	Dictionary	Frequency
w4w0w1w5w6	w4w0	Words available?		
w4w0w1w5w6	w4w0	Add to dictionary		
w4w0w1w5w6	w4w0	Pattern exists?		
w4w0w1w5w6	w4w0w1	Add word to pattern		
w4w0w1w5w6	w4w0w1	Words available?		
w4w0w1w5w6	w4w0w1	Add to dictionary	w4w0w1	1
w4w0w1w5w6	w4w0w1	Pattern exists?		
w0w1w5w6	w0w1	Take new pattern		
w0w1w5w6	w0w1	Words available?		
w0w1w5w6	w0w1	Add to dictionary		
w0w1w5w6	w0w1	Pattern exists?		
w0w1w5w6	w0w1w5	Add word to pattern		
w0w1w5w6	w0w1w5	Words available?		
w0w1w5w6	w0w1w5	Add to dictionary		
w0w1w5w6	w0w1w5	Pattern exists?		
w0w1w5w6	w0w1w5w6	Add word to pattern		
w0w1w5w6	w0w1w5w6	Words available?		
w0w1w5w6	w0w1w5w6	Add to dictionary	w0w1w5w6	1
w0w1w5w6	w0w1w5w6	Pattern exists?		
w1w5w6	w1w5	Take new pattern		
w1w5w6	w1w5	Add to dictionary		
w1w5w6	w1w5	Pattern exists?		
w1w5w6	w1w5w6	Add word to pattern		
w1w5w6	w1w5w6	Words available?		
w1w5w6	w1w5w6	Add to dictionary	w1w5w6	1
w1w5w6	w1w5w6	Pattern exists?		
w5w6	w5w6	Take new pattern		
w5w6	w5w6	Words available?		
w5w6	w5w6	Add to dictionary		
w5w6	w5w6	Pattern exists?		

Input Sequence	Pattern	Step	Dictionary	Frequency
		Take new pattern		
		Words available?		
		Exit		

Figure6:

Algorithm Pseudo Code

Hide Copy Code

```
ConstructPatterns(src, delimiters, n, fixed)
{
    des = AllocateBuffer()
    Copy(des, src)
    DiscardDelimiters(des, delimiters)
    dic = InitializePatternsDictionary()

    pattern = InitializeNewPattern(des)
    While(des)
    {
        node = dic.Insert(pattern)
        if(!fixed AND node.IsRepeated)
            AddWordToPattern(des, pattern)
        else
            pattern = InitializeNewPattern(des)
        UpdateBuffer(des)
    }
}
```

Code Description

```
#include <iostream>
#include <vector>
#include <string>
#include <fstream>

int main()
{
	std::vector<std::string> dict;
	std::vector<std::string>::iterator it;
	std::ifstream inf;
	std::string temp, word, sentence;
	bool done = false;
	char input;

	inf.open("dict.txt");

	while(!inf.eof())
	{
		inf >> word;
		dict.push_back(word);
	}

	temp = "";
	word = "";
```

```
	sentence = "";

	std::cout << "Press 1 to accept autocomplete word\n" <<
			"Press 2 to accept current word\n" << std::endl;

	while(!done)
	{
		std::cin >> input;
		if(input == '1')
		{
			sentence += temp + ' ';
			word = "";
			std::cout << sentence << std::endl;
		}
		else if(input == '2')
		{
			sentence += word + ' ';
			word = "";
			std::cout << sentence << std::endl;
		}
		else if(input != '0')
		{
			word += input;
			for(it = dict.begin(); it != dict.end(); ++it)
			{
				if(word == (*it).substr(0, word.length()))
				{
					std::cout << (*it) << std::endl;
					temp = (*it);
					break;
				}
			}
		}
		else
			done = true;
	}

	std::cout << sentence << std::endl;

	return 0;
}
```

GRAPHS AND ANALYSIS(x-axis:frequency of words,y-axis:probability percentage)

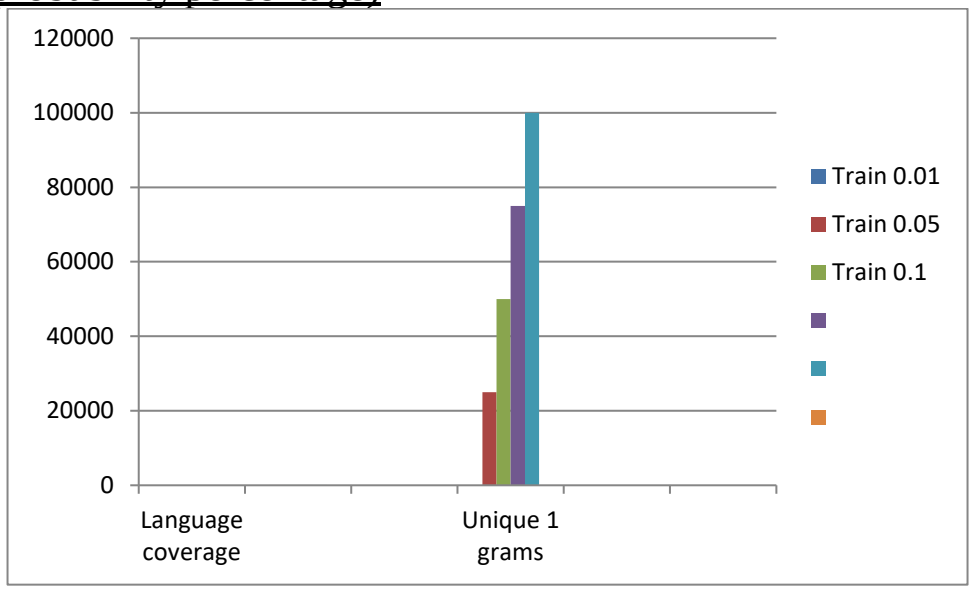


Figure7:

Outputs observed:

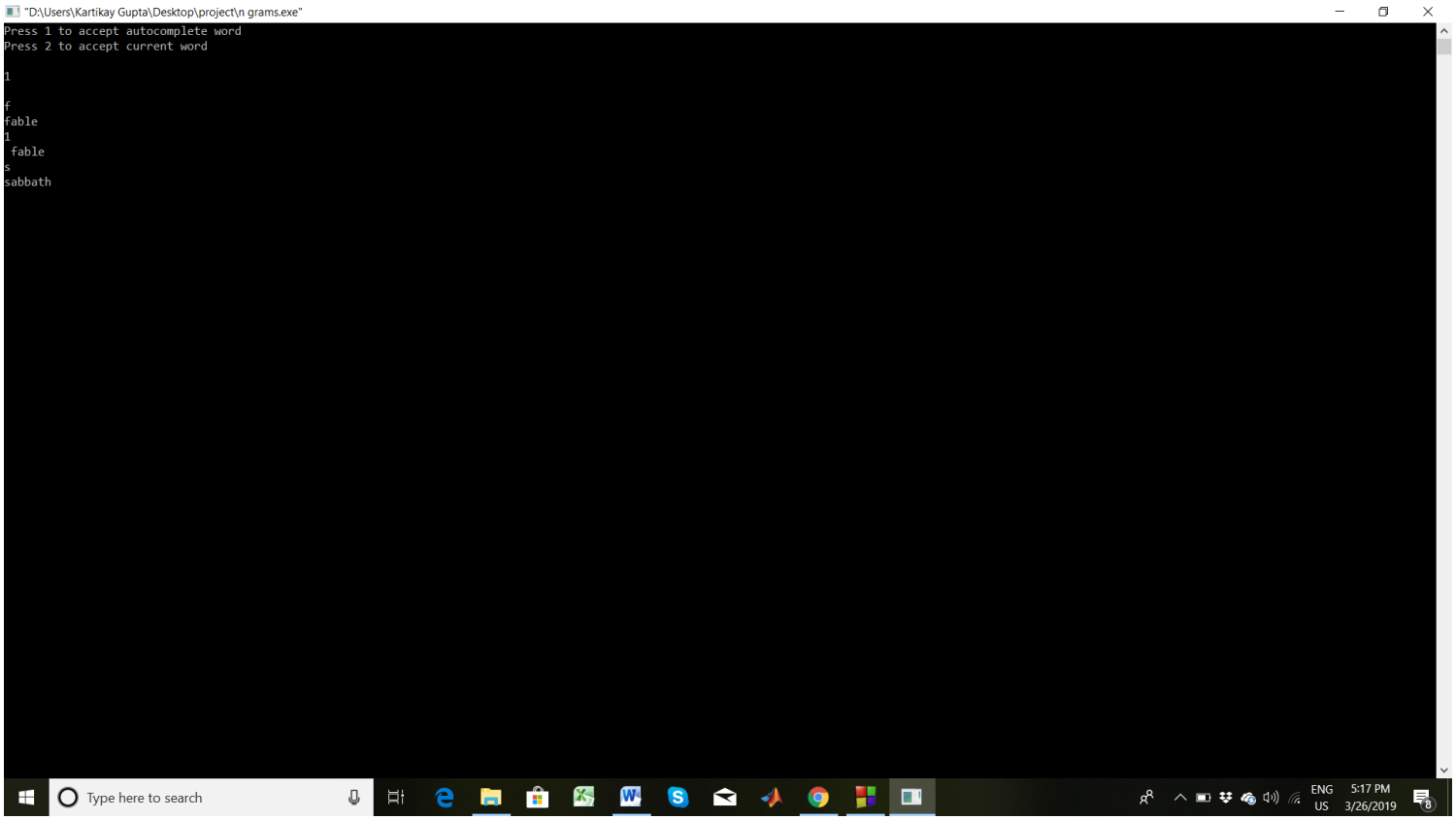


Figure 8:

CONCLUSION:

Word Predictor have application in messaging application like watsapp, web search engines, word processors, command like interpreters etc. The original purpose of word prediction software was to help people with physical disabilities increase their typing speed, as well as to help them decrease the number of keystrokes needed in order to complete a word or a sentence.[17]

Thus in this front we developed our own program for word predictor using data structure trie which definitely increases efficiency of the user by at least 10%.

So, while comparing the two approaches i.e. data structure TRIE and N-grams we found that the graphs of both are depicting the word prediction separately.[18]

While seeing the outputs of both they both were approximately same , but due time complexity and other features we can say that data structure TRIE is a better option for text/word prediction.[19]

REFRENCES:

1) Todman, J., & Dugard, P. (2001). Single-case and small-n experimental designs: A practical guide to randomization tests. Mahwah, NJ: Lawrence Erlbaum Associates.

2) Tumlin, J., & Heller, K. (2004). Using word prediction software to increase typing fluency with students with physical disabilities. Journal of Special Education Technology, 19(3). Retrieved September 24th, 2006 from http://jset.unlv.edu/19.3/tumlin/first.html

3) Weller, H. G. (1996). Assessing the impact of computer-based learning in science. Journal of Research on Computing in Education, 28, 461- 485.

4) Williams, S. (2002). How speech-feedback and word prediction software can help Students write. TEACHING Exceptional Children, 34, 72-78.

5) Zhang, Y. (2000). Technology and the writing skills of students with learning disabilities. Journal of Research on Computing in Education, 32, 467-478.

6) Edyburn, D. L. (2005). Assistive technology and students with mild disabilities: From consideration to outcome measurement. In D.L. Edyburn, K. Higgins, & R. Boone (Eds.), Handbook of special education technology research and practice (pp. 239-270). Whitefish Bay, WI: Knowledge by Design, Inc.

7) Edyburn, D. L. (2001). Critical issues in special education technology research: What do we know? What do we need to know? In M. Mastropieri, & T. Scruggs, (Eds.), Advances in learning and behavioral disabilities, Vol. 15, NY: JAI Press, pp. 95-118.

8) Handley-More, D. (2003). Facilitating written word using computer word processing and word prediction. American Journal of Occupational Therapy, 57(2), 139-151.

9) Higgins, E. L. & Raskind, M. H. (2000). Speaking to read: The effects of continuous vs. discrete speech recognition systems on the reading and spelling of children with learning disabilities. Journal of Special Education Technology, 15(1), 19-30.

10) Higgins, E. L., & Raskind, M. H. (2004). Speech recognition- based and automaticity programs to help students with severe reading and spelling problems. Annals of Dyslexia, 54(2), 173-177.

11) Horner, R. H., Carr, E. G., Halle, J., McGee, G., Odom, S., & Wolery, M. (2005). The use of single-subject research to identify evidence-based practice in special education. Exceptional Children, 71, 165-179.

12) Lewis, R. B., Graves, A. W., Ashton, T. M., & Kieley, C. L. (1998). Word processing tools for students with learning disabilities: A comparison of strategies to increase text entry speed. Learning Disabilities Research & Practice, 13, 95-108.

13) MacArthur, C. (1998). Word processing with speech synthesis and word prediction: Effects on the dialogue journal writing of Students with learning disabilities. Learning Disabilities Quarterly, 21, 151-166.

14) Parette, H. P., Wojcik, B. W., Peterson-Karlan, G., & Hourcade. J. J. (2005). Assistive technology for Students with mild disabilities: What's cool and what's not. Education and Training in Developmental Disabilities, 40, 320-331.

15) Quinlan, T. (2004). Speech recognition technology and students with writing difficulties: Improving fluency. Journal of Educational Psychology, 96, 337-346.

16) Raskind, M. H., & Higgins, E. H. (1995). The effects of speech synthesis on proofreading efficiency of postsecondary students with learning disabilities. Learning Disability Quarterly, 18, 141-158.

17) Raskind, M. H. & Higgins, E. L. (1999). Speaking to read: The effects of speech recognition technology on the reading and spelling performance of children with learning disabilities. Annals of Dyslexia, 49, 251-281.

18) Reagan, K. S., Mastropieri, M. A., & Scruggs, T. E. (2005). Promoting expressive writing among students with emotional and behavioral disturbances via dialogue journals. Behavioral Disorders, 31, 35-52.

19) Scherer, M. J. (2005). Living in the state of stuck: How assistive technology impacts the lives of people with disabilities, (4th ed.). Brookline, MA: Brookline Books