Efficient tree construction for formal language query processing

K.B. Madhuri† **M. Shashi**†† **and P.G. Krishna Mohan**†††,

†Gayatri Vidya Parishad, College of Engineering, Visakhapatnam, Andhra Pradesh, India
††Andhra University , Visakhapatnam, Andhra Pradesh, India
†††JNT University, Hyderabad, Andhra Pradesh, India

Summary
This paper describes the construction of a tree for a given database of strings for formal language query processing. A query can be presented in the form of a Regular Expression (RE) or a Context-Free Grammar (CFG). A special structure for representing the query which can be used for efficient searching is also described. This special structure is a parse tree in the case of a regular expression and Greibach normal form in the case of a context-free grammar. The proposed algorithms are a preprocessing step for search algorithms which bypass the construction of a separate automaton for a given query.

Key words:
n–ary tree, Regular Expression, Parse Tree, Context-Free Grammar, Greibach normal form.

1. Introduction

In computing, a regular expression, often called a pattern, is an expression that describes a set of strings, according to certain syntax rules. The need to search for regular expressions arises in many text based applications, such as document retrieval, text editing and computational biology. Regular expressions are used by many text editors and utilities to search and manipulate bodies of text based on certain patterns. Many programming languages support regular expressions for string manipulation. For example, Perl and Tcl have a powerful regular expression engine built directly into their syntax. The set of utilities (including the editor ed and the filter grep) provided by Unix distributions were the first to popularize the concept of regular expressions.

Context-free grammars are powerful enough to describe the syntax of most programming languages; in fact, the syntax of most programming languages is specified using context-free grammars. On the other hand, context-free grammars are simple enough to allow the construction of efficient parsing algorithms which, for a given string, determine whether and how it can be generated from the grammar.

Most existing literature for searching a regular expression are usually done by converting the regular expression into a deterministic finite automaton [4,16]. The present work consists of two phases. In first phase, all the elements present in the database are stored in a tree. The structure of the tree is similar to n-ary tree where n= $|\Sigma|$, the number of elements of fixed alphabet set. Each node of the tree except root node corresponds to the elements of alphabet set Σ. A technique to reduce the height of the tree is also proposed. In second phase, algorithms for construction of parse tree in the case of a regular expression and Greibach normal form in the case of a context-free grammar are described.

The rest of the paper is organized as follows. In section 2, we discuss some of the previous work. Section 3 presents an algorithm for construction of tree for a given set of strings and the algorithm for reducing the height of the tree. Section 4 contains the definition of regular expression and construction of the parse tree for a given regular expression. Section 5 describes the definition of context free grammar and conversion of given grammar to Greibach Normal Form. Section 6 contains comparative study. Section 7 presents conclusion and future work.

2. Previous Work

Suffix trees are used extensively for different string processing problems. Linear time algorithms for constructing efficient suffix trees have been suggested by Weiner[19], McCreight[14] and Ukkonen[18]. Folga et al.[9] proposed q-gram matching algorithm which uses a tree data structure similar to trie to store all q-grams present in the text. The number of nodes in the tree increases with the number of unique substrings in the text and with the tree depth. They proposed a tree redundancy pruning algorithm to reduce the size of the tree. The algorithm also uses suffix links for efficient runtime substring matching. Bedathur et al.[3] described a buffering strategy, called TOP-Q which improves the performance of the Ukkonen's algorithm(which uses

Manuscript received March 5, 2007
Manuscript revised March 25, 2007

suffix links) when constructing on-disk suffix trees. Several $O(n^2)$ and $O(n \log n)$ algorithms for constructing suffix trees are described in [11].

Hunt et al.[13] suggested a different strategy where the authors drop the use of suffix links and use $O(n^2)$ algorithm with a better locality of reference. Suffix arrays have also been used as an alternative to suffix trees for specific string matching algorithms [6,8,15]. Burkhardt et al.[5] proposed QUASAR which uses a suffix array to retrieve the positions of any given q-grams in the text. The preprocessing step of QUASAR takes $O(n \log n)$ time.

Aho and Corasick[1] proposed a linear time algorithm with multiple strings which converts strings into Deterministic Finite Automaton(DFA). The algorithm takes linear time to cover the entire length of the string. Coit et al.[7] presented a fast string matching algorithm which stores the strings in a tree similar to Aho and Corasick. Navarro and Ranot[16] described a technique for compact deterministic finite automaton representation based on the properties of Glushkov's NFA construction. Thompson[17] described an algorithm to search a regular expression of length m in a text of length n is to convert the expression into a non-deterministic finite automaton (NFA) with O(m) nodes. There is a simple algorithm described by Aho et al.[2] which constructs NFA from a given regular expression, R that accepts L(R) in $O(|r|)$ time. Hopcroft and Ullman[12] described an algorithm to convert NFA to a NFA without ε- transitions $(O|r|^2)$ states and to a DFA($O(2^{|r|}$) states in the worst case).

3. Construction of a tree for a given database

The tree structure provides a general framework for a systematic study of stream data. Trees are computationally efficient for storage, addition, and searching for sets of strings. A database can be efficiently represented using a tree. We construct a tree which is similar to n-ary tree where $n=|\sum|$, the no. of elements of fixed alphabet.

3.1 Node Structure

Each node of the tree has the following fields.

i) Information Field:
   This field contains information in the form of characters.

ii) Pointers to the child nodes:
   The pointers that hold the addresses of child nodes (the no of children is at max is n).

iii) Flag :
   Flag is used to recognize whether the particular string obtained by concatenating the strings from the root up to the current node in the left to right sequence is present as an element in the database or not. The flag of a node is 1 if and only if the string upto the corresponding node is contained in the database. The node structure is given in Fig 1.

```
struct node  /* Structure of node */
{      char *info;
       struct item **child;
       int *flag;
}
```

Information	child 1-ptr	child 2-ptr	…..	child n-ptr	flag

Fig.1. Node Structure

3.2 Construction of a tree

Let us consider an input alphabet $\sum = \{x, y\}$ and the database strings 'xy' and 'yx' are taken. The construction of the tree begins by creating a root node. Then the strings are added one after the other. The pictorial representation of initial tree is given in fig 2. The algorithm for construction of a tree is given in fig. 3.

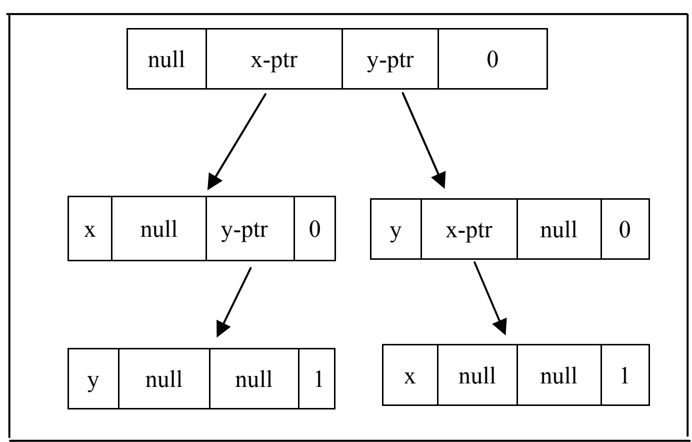

Fig. 2: Construction of initial tree for the strings "xy" and "yx"

```
Procedure build_tree(char data[],struct item *p)
#define NOT_A_SYMBOL -10
//Data is the database string
//*p is the Tree Pointer
{
      for all the characters in the string data
     {
      k=position of current character in the alphabet, if character
      is not present in the alphabet then store
      NOT_A_SYMBOL;
        if(k==NOT_A_SYMBOL)
        Print current character is not in the alphabet and return

          if(p->child[k]==null)
                  {
                  p->child[k]=create new node;
                   p=p->child[k];
                   Allocate memory to information field
                   p->info[0]=data[i];    /* Data[i] is the ith character
                                                   in the string data */
                   Allocate memory to boolean field
                   p->flag[0]=0;
                   Initialize the child node addresses of node p to
                  null
                  }
         else
                  p=p->child[k];
     }
                 p->flag[0]=1;
}
```

Fig. 3 : Algorithm for construction of a tree

```
void decrease_db_tree(struct item *p)
 // Input : tree from algorithm (fig. 3)
 // Output :  tree after reducing its height
 // *p is the current pointer in the tree
{
    outdegree=1
    While (outdegree==1)
     {
       outdegree=0
       for all the child node addresses find the number of child
       node pointers that do not have   NULL  and store that
       value in outdegree
       if(outdegree==1)
       {
            merge child node to the parent node
          // append its information field
         // append its flag
         /* update the child node addresses of the parent with
            that of child */
       }
       else if (outdegree > 1)
        {
          for all the child node pointers that are not NULL
            call  decrease_db_tree(p→child[i] )
        }
             else return ;
    }
 }
```

Fig. 4: Algorithm for reducing the height of the tree

3.3 The algorithm for reducing the height of the tree

The maximum height of the tree will be the length of the longest string contained in the database. Outdegree of a node means the number of child nodes coming from that particular node. If the outdegree of a parent node is 1 then the child node can be merged to parent node. The process of merging is as follows: The child node's information field is appended with the current node's information field. The child node's flag is appended with the current node's flag. The child pointers of the current node are replaced with the child pointers of the current node's child node. By reducing the height of the database tree the number of function calls for searching the database tree is reduced in case of database containing long strings.

In the previous example (fig. 2) the outdegree of nodes at level 1 is one so they can be combined along with the child nodes so as to yield the tree with decreased height as in fig. 5. The algorithm for reducing the height of a tree is given in fig. 4.

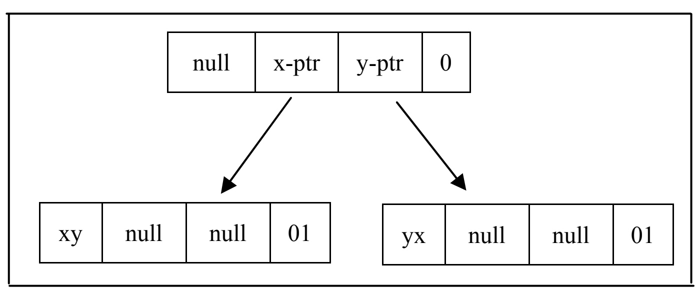

Fig. 5: Tree for the strings "xy" and "yx" after reducing the height

4. Regular Expressions

4.1 Definition of Regular Expression

Regular expressions can be expressed in terms of formal language theory. Regular expressions consist of constants and operators that denote sets of strings and operations over these sets, respectively. Given a finite alphabet Σ, the following constants are defined:

- (*empty set*) Ø denoting the set Ø
- (*empty string*) ε denoting the set {ε}
- (*literal character*) *a* in Σ denoting the set {*a*} and the following operations:
- (*concatenation*) *RS* denoting the set { αβ | α in *R* and β in *S* }. For example {"ab", "c"}{"d", "ef"} = {"abd", "abef", "cd", "cef"}.
- (*alternation*) *R*|*S* denoting the set union of *R* and *S*.
- (*Kleene star*) *R** denoting the smallest superset of *R* that contains ε and is closed under string concatenation. This is the set of all strings that can be made by concatenating zero or more strings in *R*.

For example, {"ab", "c"}* = {ε, "ab", "c", "abab", "abc", "cab", "cc", "ababab", ... }.

The above constants and operators form Kleene algebra.

Examples

a|b* denotes {ε, *a*, *b*, *bb*, *bbb*, ...}

(a|b)* denotes the set of all strings consisting of any number of *a* and *b* symbols, including the empty string b*(a|b*)* the same

ab*(c|ε) denotes the set of strings starting with *a*, then zero or more *b*s and finally optionally a *c*.

(aa|ab(bb)*ba)*(b|ab(bb)*a)(a(bb)*a|(b|a(bb)*ba)(a a|ab(bb)*ba)*(b|ab(bb)*a))* denotes the set of all strings which contain an even number of '*a*' s and an odd number of '*b*' s.

4.2 Construction of Parse Tree

Regular Expression is taken as input and binary parse tree is constructed. It is of the following form shown in the fig 6.

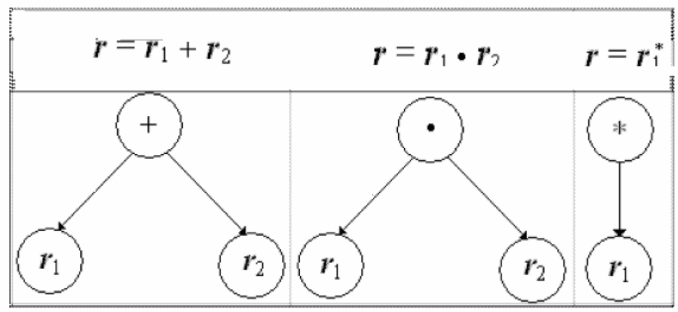

Fig. 6. Parse tree representation of regular expression

The operators + (Union) and . (Concatenation) will have two children and * (Kleene Star) will have one left child only. The algorithm for construction of a parse tree for a given regular expression is given in fig. 7.

```
Struct parse_tree_node  /* Parse Tree node structure */
{
   Char *dat;
   Struct parse_tree_node *left;
   Struct parse_tree_node *right;
}

Input : Regular Expression is taken
Output: Parse Tree Representation of Regular Expression
          Algorithm:
Step 1 : Scan the RE from left to right and check whether all
          the symbols present in the RE match with the symbols
          of fixed alphabet.
Step 2 : Scan the RE from left to right and check whether the
          brackets are matched or not
Step 3:  If it is no in any of the  above two cases then it means
          user entered wrong RE and  go to Step 5   Else Go to
          Step 4
Step 4 :  Initialize a global variable top=0
           Create a global array 'a' to store the nodes of parse
           tree
            Scan the entire RE from left to right
```

```
            If current character is not ')'
                    If current character is '('
                            Store '(' in temporary string
                     else if current character is an alphabet
                            Store the part of RE from current
                            character till it encounters an operator in
                            some temporary string
                     else if current character is an operator
                            Store the operator in temporary string
                     end if
                     call insert()
             else
                    call delete()
             end if
end
Step 5: Stop

void insert()
{
              node1=create a new node of parse tree
              node1->dat=Allocate memory equal to the size of
             temporary string obtained from  the main algorithm
              node1->left=NULL;
              node1->right=NULL;
              a[top++]= node1;
}
void delete1()
{            node2=a[top-1];
             node2->left=a[top-2];
             node2->right=NULL;
             top=top-3;
             a[top++]=node2;
}
void delete()
{
 if (operator which is stored in temporary string is '*') then
                          delete1();
else if(operator which is stored in temporary string is'+'or '.')
then
             {            node2=a[top-2];
                          node2->left= a[top-3];
                          node2->right=a[top-1];
                          top=top-4;
                          a[top++]=node2;
             }

 end if

}
```

Fig. 7: Algorithm for construction of a parse tree for a given regular expression

5 Context-Free Grammar (CFG)

5.1 Definition of CFG

Just as any formal grammar, a Context-Free Grammar G can be defined as a 4-tuple:

$G = (V_t, V_n, P, S)$ where

- V_t is a finite set of terminals
- V_n is a finite set of non-terminals
- P is a finite set of production rules
- S is an element of V_n, the distinguished starting non-terminal.
- elements of P are of the form
  $V_n \longrightarrow (V_t \cup V_n)^*$

A language L is said to be a Context-Free Language (CFL) if and only if there is a Context-Free Grammar G such that L=L(G). More precisely, it is a language whose words, sentences and phrases are made of symbols and words from a Context-Free Grammar.

Example

The grammar G = ({S}, {a, b}, S, P), with productions

$$S \rightarrow aSa,$$
$$S \rightarrow bSb,$$
$$S \rightarrow \epsilon,$$

is context-free. A typical derivation in this grammar is

S => aSa =>aaSaa => aabSbaa => aabbaa

This makes it clear that the Language obtained from the above grammar is of the following form

$$L(G) = \{ww^R : w \in \{a, b\}^*\}.$$

5.2 Steps for conversion of given context-free grammar into Greibach normal form

Conversion of given grammar into Greibach normal form is done by the following steps

Step1 : Elimination of Useless symbols
Step2 : Elimination of null productions
Step3 : Elimination of Unit Productions
Step4 : Conversion to Chomsky like normal form
Step5 : Conversion to Greibach normal form

5.2.1 Elimination of Useless symbols

Let G=(V_n, V_t, P, S) be a Grammar. A symbol X is useful if there is a derivation S $\overset{*}{\Rightarrow}$ α Xβ $\overset{*}{\Rightarrow}$ w for some α, β and w, where w is in V_t^* . Otherwise X is uselessness. There are two aspects to usefulness. First some terminal string must be derived from X and second, X must occur in some string derivable from S. These two conditions are not, however, sufficient to guarantee that X is useful, since X may occur only in sentential forms that contain a non terminal form which no terminal string can be derived. The algorithm for elimination of useless symbols is described in fig. 8 which is described in [11].

```
oldv =Ø
newv={A| A → w for some w in Vt* }
while (oldv != newv)
  {
   oldv= newv
   newv=oldvỤ{A| A→ β for some β  in (Vt Ụ oldv) * }
 }
 Vn’ = newv
```

Fig. 8 : Algorithm for Elimination of useless symbols

Example

Consider G = ({S, A, B}, {a,b}, P, S) with productions

S → A,
A → aA | ε,
B → bA,

The non terminal B is useless and so is the production B → bA. Although B can derive a terminal string, there is no way we can achieve S$\overset{*}{\Rightarrow}$ xBy.

5.2.2. Elimination of null productions

One kind of production that is sometimes undesirable is one in which the right side is the empty string.

Any production of a context free grammar of the form A → ε is called a null production.

Any non-terminal A for which the derivation A $\overset{*}{\Rightarrow}$ ε is possible is called nullable.

The algorithm for elimination of null productions is described in fig. 9.

The set V_n of all nullable non-terminals of G is found, using the following steps:

Step 1: For all productions A→ ε, put A into V_n
Step 2: Repeat the following step until no further non-terminals are added to V_n

For all productions
B → $A_1A_2A_3…A_n$
Where A_1, A_2,…A_n are in V_n, put B into V_n.

Once there set V_n has been found, we are ready to construct P’. To do so, we look at all the productions in P of the form A → $x_1x_2…x_m$, where m>=1,

where each x_i Є (V_n U V_t). For each such production of P, we put into P’ that production as well as all those generated by replacing nullable variables with ε in all possible combinations. For example, if x_i and x_j are both nullable, there will be one production in P’ with both xi replaced with ε, one in which x_j is replaced with ε, and one in which both x_i and x_j are replaced with ε. There is one exception: if all x_i‘s are nullable, the production A → ε is not put into P’.

Fig. 9 : Algorithm for Elimination of null productions

5.2.3 Elimination of Unit Productions

Definition : Any production of a context free grammar of the form A → B, where A,B Є V_n is called a Unit Production.

To eliminate the unit productions we proceed as follows Given a CFG G = (V_n, V_t,P,S), construct CFG G1 = (V_n, V_t, P’,S): The algorithm for elimination of unit productions is explained in fig. 10.

1. Include all nonunit productions of P into a new set of productions P’
2. Suppose that A $\overset{*}{\Rightarrow}$ B, for A,B in V_n
3. Add to P’all the productions of the form A→α , where B→α is a non unit production of B

Fig. 10 : Algorithm for Elimination of Unit productions

5.2.4 Conversion to Chomsky Like normal form

Given grammar is said to be in Chomsky normal form(CNF) if for every production in the grammar is as follows:

$$A \rightarrow BC$$
or
$$A \rightarrow a$$

Where, 'A', 'B' & 'C' are any non terminals and 'a' is any terminal.

However, by converting to CNF, the numbers of productions are increased leading to performance degradation.

So, we have made a minor change here and converted the grammar into the following form:

$$A \rightarrow X$$
or
$$A \rightarrow a$$

Where, 'A' is any non terminal, 'a' is any terminal and 'X' is a sequence of non-terminals. The algorithm for conversion to Chomsky Like normal form is given in fig. 11.

Construct a new set of productions P' from P as follows

1. All the productions in P of the form $A \rightarrow a$ and $A \rightarrow X$ are included
2. Consider $A \rightarrow x_1\, x_2\, x_3\, x_4\, x \ldots\ldots x_n$ where each x_i may be a terminal or non terminal. If x_i is a terminal then add a new production $C \rightarrow x_i$ to P' and C to V_n'. Replace x_i with C in all productions.
3. Repeat the for all remaining productions.

Fig. 11 : Conversion to CNF equivalent

Example

Given grammar with productions

$S \rightarrow CDa$,
$C \rightarrow aab$,
$D \rightarrow Cc$,

Equivalent grammar to Chomsky Like normal form is

$S \rightarrow CDB_a$,
$C \rightarrow B_aB_aB_b$,
$D \rightarrow CB_c$,
$B_a \rightarrow a$,
$B_b \rightarrow b$,
$B_c \rightarrow c$.

5.2.5 Conversion to Greibach normal form

In Greibach normal form [10], we put restrictions not on the length of the right sides of the production, but on the productions in which terminals and nonterminals appear.

Definition : A context-free grammar is said to be in Greibach normal form if all the productions have the form

$$A \rightarrow aX,$$
Where $a \in V_t$ and $X \in V_n^*$

The algorithm described below (fig. 12) converts a grammar presented as a list of production rules into its Greibach normal form.

Step1: Eliminate all useless symbols, null productions and unit productions
Step2: Convert the grammar into Chomsky like normal form.
Step3: Rename all the non terminals as follows
If S,A,B……are non terminals, then they are renamed as A_1,A_2,A_3…..respectively.
Step4: If a production such that the subscript of non terminal on left hand side is greater than or equal to the subscript of first non terminal on RHS is present then apply Lemma1 or Lemma2 respectively.
Refer Fig. 13 for Lemma1 and Fig. 14 for Lemma2.

Fig. 12 : Conversion to Greibach normal form

Let, $G=\{V_n,V_t,P,S\}$ be CFG.
Let, $A \rightarrow BX$ be a production in P where X be a set of non terminals and $B \rightarrow \beta_1 \mid \beta_2, \mid \beta_3 \ldots \mid \beta_s$ be the set of all B productions. Where β_i can have the following forms
(i) Sequence of non terminals
(ii) A terminal
Define $P1 = (P - \{A \rightarrow B\gamma\}) \cup \{ A \rightarrow \gamma\, \beta_i \mid 1 <= i <= s\}$
Then $G_1 =\{ V_n, V_t,P_1,S\}$ is a context free grammar equivalent to G.

Fig. 13: Lemma 1

Let, G={ V_n, V_t,P,S} be given CFG.
Let, $A \rightarrow AX_1 \mid AX_2 \mid AX_3 \mid AX_4, \ldots\ldots, \mid AX_n$
Where Xn is the set of non terminals and $A \rightarrow a_1 \mid a_2 \mid a_3 \mid a_4 \mid \ldots\ldots \mid a_m$ be the remaining productions. Where a_i, 1 <= i <= m is a terminal.

Let G′={ V_n U {Z},V_t,P′,S} be the CFG formed by adding the variable Z to Vn and replacing all A-productions by the following productions.

1. $A \rightarrow a_i$
   $A \rightarrow a_i Z$
   For $1 \le i \le m$
2. $Z \rightarrow X_i$
   $Z \rightarrow X_i Z$
   For $1 \le i \le n$

Then L(G') = L(G)

Fig. 14: Lemma 2

6. Comparative Study

In our search algorithms query string is presented in the form of regular expression and context free grammars. If the query is a regular expression the main problems with the traditional approaches are described below. There is a simple algorithm described by Aho et al.[2] which constructs a NFA from a given regular expression that accepts L(R) in O(|r|) time. Hopcroft and Ullman [12] described an algorithm to convert NFA to a NFA without epsilon transitions ($O|r|^2$) states and to a DFA($O(2^{|r|})$) states in the worst case).Thus if DFA is followed both the construction time and number of states may become exponential in the size of Regular Expression. In our proposed algorithm we bypass the construction of DFA there by eliminating the above mentioned problem and we have constructed the parse tree whose height will increase if we give a regular expression of large size.

7. Conclusion and future work

In this paper a tree is constructed similar to n-ary tree where n = |Σ|, the number of elements of fixed alphabet for a formal language query processing. An algorithm is described to reduce the height of the tree. A query can be presented in the form of a Regular Expression (RE) or a Context-Free Grammar (CFG). A binary parse tree is constructed if query is presented in the form of a regular expression and Greibach normal form is described in the case of a context-free grammar. In future work, we are developing search algorithms that use these representations. The algorithms bypass construction of a separate automaton for a given query.

Acknowledgments

The Authors are thankful to Dr. D. Ravi, Ph.D (IISc, Bangalore), Professor in Computer Science & Engineering, Gayatri Vidya Parishad College of Engineering for his valuable guidance and constant support in completing this work. The assistance of G.Phanindra, K.V.V.N.Ravi Shankar and P.Deepak Sreenivas in the implementation of algorithms was appreciated.

References

[1] A.V. Aho and M.J. Corasick, "Efficient String Matching: An Aid to Biblographic Search", *Communications of the ACM*, vol. 18, no. 6, pp. 333-340, 1975.
[2] A.V. Aho, J.E. Hopcraft and J.D. Ullman, "The Design and Analysis of Computer Algorithms", *Addison-Wesley, Reading, Mass.*, 1974.
[3] S.J. Bedathur and J.R. Harista, "Engineering a Fast Online Persistent Suffix Tree Construction", ICDE, pp. 720-731, 2004.
[4] R.A. Baeza-Yates, "Fast Text Searching for Regular Expression or Automaton Searching on Tries", *Journal of the ACM*, vol.43, no. 6, pp. 915-936, 1996.
[5] S. Burkhardt, A. Crauser, P. Ferragina, H. Lenhof, E. Rivals and M. Vingron, "q-gram based database searching using a suffix array(QUASAR)", *Proc. of the Third Ann. Int'l Conf. on Research in Computational Molecular Biology*, pp. 77-83, 1999.
[6] L.L. Cheng, D.W.L. Cheung and S.M. Yiu, "Approximate String Matching in DNA Sequences", *Proc. of the Eighth International Conference on Database Systems for Advanced Applications*, pp. 303-310, 2003.
[7] J. Coit, S. Staniford and J. McAlerney, "Towards Faster String Matching for Intrusion Detection or Exceeding the speed of Snort", *Proc. DARPA Information Survivability Conf. and Exposition(DISCEX II '02)*, vol. 1, pp. 367-373, 2001.
[8] A. Crauser and P. Ferragina, "A Theoretical and Experimental Study on the Construction of Suffix Arrays in External Memory", *Algorithmica*, 32(1):1-35, 2002.
[9] P. Folga and W. Lee, "q-Gram Matching Using Tree Models", *IEEE Trans. Knowledge and Data Engg.* Vol. 18, No. 4, pp. 433-447, 2006.
[10] S.A. Greibach, "A New Normal-Form Theorem for Context-Free Phrase Structure Grammars", JACM, Vol. 12, No. 1, pp. 42-52, 1965.
[11] D. Gusfield, "Algorithms on Strings, Trees and Sequences - Computer Science and Computational

Biology", *Cambridge University Press*, 1997.

[12] J.E. Hopcroft and J.D. Ullman, "Introduction to Automata Theory, Languages and Computation", *Addison-Wesley, Reading Mass.,* 1979.

[13] E. Hunt, M.P. Atkinson and R.W. Irving, "A Database Index to Large Biological sequences", The VLDB J. 7(3), pp. 139-148, 2001.

[14] E.M. McCreight, "A Space-Economical Suffix Tree Construction Algorithm", *J.ACM*, vol. 23, no. 2, pp. 262-272, 1976.

[15] G. Navarro, R.A. Baeza-Yates and J. Tarhio, "Indexing Methods for Approximate String Matching", *IEEE Data Engineering Bulletin*, 24(4):19-27, 2001.

[16] G. Navarro and M. Raffinot, "Compact DFA Representation for Fast Regular Expression Search", *Algorithm Engineering*, pp. 1-12, 2001.

[17] K. Thomson, "Regular Expression Search Algorithm", *communications of the ACM*, 11(6): pp. 419-422, 1968.

[18] E. Ukkonen, "On-Line Construction of Suffix Trees", *Algorithmica*, vol. 14(3), pp. 249-260, 1995.

[19] P. Weiner, "Linear Pattern Matching Algorithms", *Proc. of the 14th Annual Symposium on Foundations of Computer Science*, *IEEE Computer Society,* pp. 1-11, 1973.

K.B.Madhuri received the M.Sc degree in Applied Mathematics from Sri Venkateswara University in 1992 and also she got M.Tech. degree in Computer Science and Technology from Andhra University in 1999. She is pursing Ph.D from Jawaharlal Nehru Technological University. Presently She is working as Associate Professor in Computer Science and Engineering at Gayatri Vidya Parishad, College of Engineering, Visakhapatnam, Andhra Pradesh, India. Her research interests include Data Mining and Pattern Recognition. She is an associate member of Institute of Engineers(India).

M.Shashi received her B.E. Degree in Electrical and Electronics and M.E. Degree in Computer Engineering with distinction from Andhra University. She received Ph.D in 1994 from Andhra University and got the best Ph.D thesis award. She is a professor in Computer Science and Systems Engineering at Andhra University, Andhra Pradesh, India. She received AICTE career award as young teacher in 1996. She is a co-author of the Indian Edition of text book on "Data Structures and Program Design in C" from Pearson Education Ltd. She published technical papers in National and International Journals. Her research interests include Data Mining, Artificial intelligence, Pattern Recognition and Machine Learning. She is a life member of ISTE, CSI and a fellow member of Institute of Engineers(India).

P.G.Krishna Mohan received the B.Tech Degree in Electronics and Communication Engineering and M.E. Degree in Communication Systems from IIT, Roorkee. He received Ph.D from IISc, Bangalore. His Research interests include Signal Processing and Communications. He has 19 papers published to his credit in various National/International Conferences and Journals. He had been invited to give Lectures (Kalman Filtering and its Applications) at NSTL during September 1999. At Present he is a Member of Board of Studies(BOS), JNTU in the discipline of ECE .He is also the Chairman BOS of ECE Department in Autonomous JNTU College of Engineering, Hyderabad. He has completed 3 sponsored projects aided by MHRD & AICTE. He has 28 years of Research and Teaching and research experience. He is a Fellow member of IE (India) and IETE and a Life Member of ISTE.