Algorithm Design for DFA Generation: Suffix, Prefix & Substring Patterns

D. Akhil Kumar[#1], Gaurav Deswal[*2,] Dr. Gulshan Goyal[$3]

[#1] Scholar, Department of CSE, Chandigarh College of Engineering and Technology, Chandigarh (India)
*[*2] Scholar, Department of CSE, Chandigarh College of Engineering and Technology, Chandigarh (India)*
[$3] Assistant professor, Department of CSE, Chandigarh College of Engineering and Technology, Chandigarh (India)

[#1]co16312.ccet@gmail.com
[*2]*co16317.ccet@gmail.com*
[$3]*gulshangoyal@ccet.ac.in*

Abstract— **Automata theory deals with how efficiently a problem can be solved on a model of computation using an algorithm. Deterministic finite automata (DFA) is a type of finite automata (FA) used in automata theory for recognizing regular expressions. DFA designing manually is a laborious process since no fixed mathematical formulas, methods exist for designing DFA, and it cannot handle the validations for acceptance or rejection of strings. Consequently, it is difficult to represent the DFA's transition table and graph. The present paper proposes an algorithm for automatic generation of DFA. Further, comparison with Java Formal Languages and Automata Package (JFLAP), a software tool used for designing finite automata, pushdown automata, and Turing machines. This paper helps in constructing DFA with different conditions for prefix, suffix and substring of a given string based on their characteristics, strengths and their performance is compared with JFLAP tool. The novelty of proposed work lies in its approach to design the DFA, its transition table, transition graph and string recognition and rejection.**

Keywords: Automata theory, Model of computation, DFA, Transition table, Transition graph, JFLAP, Finite automata.

I. INTRODUCTION

Kurt Friedrich Gödel Turing and Alonzo Church, in 1930, discovered that some of the mathematical problems (fundamental problems) are unsolvable by a computer. This gave the birth to computability theory [1], it helps to classify the problems as being solvable or unsolvable [2]. Automata theory is the study that deals with mathematical definitions and properties of different types of computational models such as Finite Automata, context-free grammars, Turing machines etc. Study of abstract machines and automata is known as automata theory. It is a theoretical concept in computer science and discrete mathematics.

Some basic terms used in automata theory are:

i. Alphabet (Σ), it is a finite, non-empty symbols. E.g. $\Sigma = \{0, 1\}$, the binary alphabet.
ii. Strings (w), a finite sequence of symbols chose from some alphabet Σ, normally denoted by w, x, y, z. E.g. if $\Sigma = \{0, 1\}$ then 1011 and 111 are strings.
iii. Σ^*, set of all strings over an alphabet Σ.
iv. Languages (L), set of strings all of which are chosen from some Σ^*, where Σ is alphabet.
v. Problem, given Σ, L, w (in Σ^*) check whether or not w is in L.

FA is a computational model or device that has finite amount of memory (other computational devices such as computers, laptops, calculators etc.). It contains finite no. of states independent of the input size. It is a simple machine, useful in recognizing patterns [3] within input taken from Σ. The job of an FA is to accept or reject by processing an input string (w) based on whether the pattern defined by the FA is present in the input.

FA can be either deterministic or non-deterministic [6] and helps in string recognition or rejection i.e. if by the end of input string, if the current position is the final state then the string is accepted otherwise it gets rejected.

A. DETERMINISTIC FINITE AUTOMATA

DFA can be defined as in [4]. The mandatory condition for FA to be a DFA is that each input symbol has only one transition from every state. Finite Automata (M) is mathematically stated as a 5-tuple set,

$$M=(Q, \Sigma, \delta, q_0, F)$$

Where
Q is a finite set of states.
Σ is a finite set of input symbols.
δ: $Q \times \Sigma \rightarrow Q$, transition function.
q_0 is the initial state, $q_0 \in Q$ as shown in Figure 1.
F is a set of accepting/final states $F \subseteq Q$ as shown in Figure 2.

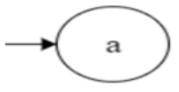
Initial State		*a, is an initial state. DFA has only one initial state*

Figure 1: Description of Initial State

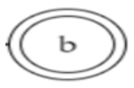
Final State		*b, is a final state. DFA has one or more final states*

Figure 2: Description of Final State

Transition function is mathematically represented as
$\delta(q, \Sigma) \rightarrow Q$, where $q \in Q$

DFA accepts a string w if starting at state q_0 the DFA ends at an accept state (F) on reading the string w. The DFA accepts a language $L \subseteq \Sigma^*$ if every string in L is accepted by M and no more which is denoted as L (M), which is pronounced as "M recognizes L".

Transitions from one internal state to another can be defined by the transition function as shown in Figure 3,

$\delta : Q \times \Sigma \rightarrow Q$

$\delta (q_i, a) \rightarrow Q$,where $q_i \in Q$ and $a \in \Sigma$

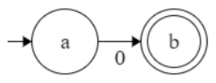
Transition		*Represented using arrow with transition symbol on it*

Figure 3: Description of Transitions

Transition Graph also called "State Transition Diagram" in which the vertices represent states and the edges represent transitions. The labels on the vertices are the names of the states while the labels on the edges are the current values of the input symbol. As shown in Figure 4, q0, q1, q2, and q3 represent states in a graph. q0 is the initial state from where transitions begin and q3 is the final state, if after string processing we are at the final state then string is accepted otherwise rejected. {a,b} represents the transition symbols through which we can go from one state to other.

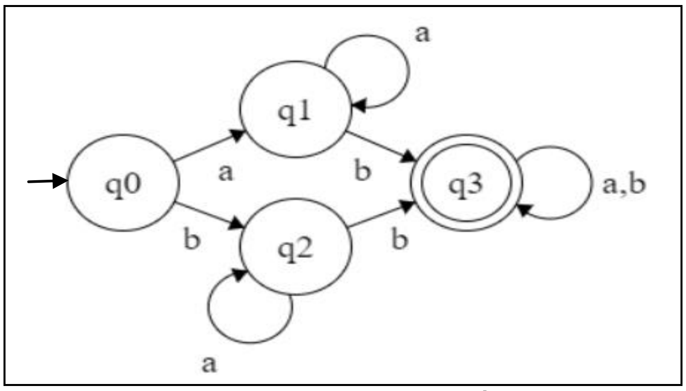


Figure 4: Transition Diagram/Graph

Transition table is a tabular representation as shown in Table 1 of the transition function that requires two arguments outputs a state. The column corresponds to the state in which the automaton will be on the input represented by that particular column. The row corresponds to the current state. The entry for one row corresponding to state q and the column corresponds to input b is the state δ(q, b). The start state is marked with an arrow and the final state is marked with double asterisk symbols, **.

Table 1: Transition Table.

states	*a*	*B*
→q0	*q1*	*q2*
q1	*q1*	*q3*
q2	*q2*	*q3*
***q3*	*q3*	*q3*

Here Q = {q0, q1, q2, q3}, Σ = {a, b} and F = {q3}

M = ({q0, q1, q2, q3}, {a, b}, δ, q0, {q3})

Transitions:

δ (q0, a) = q1	δ(q0, a) = q2
δ (q1, a) = q1	δ(q1, a) = q3
δ (q2, a) = q2	δ(q2, a) = q3
δ (q3, a) = q3	δ(q3, a) = q3

II. MOTIVATION

DFA is first and most important activity in compiler design. It is used to describe a regular expression which helps in solving logical problems such as pattern matching, path finding and speech processing [9]. However, novel learners and researchers face difficulty in designing the Deterministic finite automaton (DFA) [5] as this theoretical concept of automata theory does not have well-structured mathematical formulas or algorithms i.e. if the input problem changes then the approach to design the corresponding DFA also changes. Hence, it is the need of the hour to come up with a well-defined algorithm [7]. Researchers use JFLAP tool to handle these issues in DFA designing, but it has a major limitation that designing is manual in JFLAP. It is laborious to design a DFA manually and it does not handle validation of input alphabets and strings as well. JFLAP has no feature to draw a transition table, which is helpful in better understanding for learners. The researchers use JFLAP tools for demonstration of acceptance or rejection of a particular example with few strings.

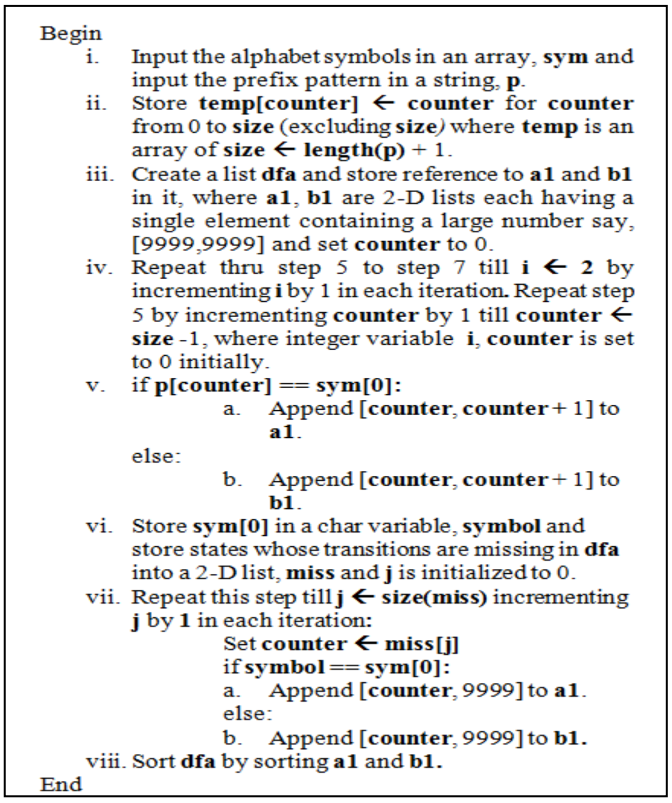
The central idea is to give researchers and beginners an algorithm [8] through which they can generate required DFAs both easily and accurately in a timely manner. The proposed algorithm will provide a transition table, transition graph, and string processing which together as a whole provides a great insight to understand and implement computation models easily. The objective of this paper is to give a well-defined algorithm to design a DFA for having suffix, prefix and substring pattern for a given string.

III. MATERIALS AND METHODS

The present section describes a simple and easily understandable approach to generate a DFA for a given prefix, suffix and substring pattern along with its transition table and transition graph. Following four subsequent sections describe the algorithms for:

A. *Algorithm 1: DFA for Prefix Pattern*

It takes the alphabet symbols as input along with the prefix pattern and outputs the DFA for any combination of input symbols having the input prefix. Following are the steps to be followed:



```
Begin
    i.    Input the alphabet symbols in an array, sym and
          input the prefix pattern in a string, p.
    ii.   Store temp[counter] ← counter for counter
          from 0 to size (excluding size) where temp is an
          array of size ← length(p) + 1.
    iii.  Create a list dfa and store reference to a1 and b1
          in it, where a1, b1 are 2-D lists each having a
          single element containing a large number say,
          [9999,9999] and set counter to 0.
    iv.   Repeat thru step 5 to step 7 till i ← 2 by
          incrementing i by 1 in each iteration. Repeat step
          5 by incrementing counter by 1 till counter ←
          size -1, where integer variable i, counter is set
          to 0 initially.
    v.    if p[counter] == sym[0]:
                  a.  Append [counter, counter + 1] to
                      a1.
          else:
                  b.  Append [counter, counter + 1] to
                      b1.
    vi.   Store sym[0] in a char variable, symbol and
          store states whose transitions are missing in dfa
          into a 2-D list, miss and j is initialized to 0.
    vii.  Repeat this step till j ← size(miss) incrementing
          j by 1 in each iteration:
                  Set counter ← miss[j]
                  if symbol == sym[0]:
                  a.  Append [counter, 9999] to a1.
                  else:
                  b.  Append [counter, 9999] to b1.
    viii. Sort dfa by sorting a1 and b1.
End
```

Figure 5: Algorithm to Generate a DFA for a Given Prefix Pattern

B. *Algorithm 2: DFA for Suffix Pattern*

It takes the alphabet symbols as input along with the suffix pattern and outputs the DFA for any combination of input symbols having the input suffix. Following algorithm uses a function **checkPattern()** that takes pattern, **p**, given string, **j** and length(pattern), size as an input and returns true if **p** equals **j** in reverse starting from index–size.

Otherwise, it returns false. Following are the steps to be followed:

```
Begin
  i.    Input the alphabet symbols in an array, sym and input the
        suffix pattern in a string, p and store references to a1, b1
        in a list, dfa where a1, b1 are 2-D lists. Also, store
        temp[counter] $\leftarrow$ counter for counter from 0 to size
        (excluding size) where temp is an array of size $\leftarrow$
        length(p) + 1.
  ii.   Create an array string_list. Repeat the subsequent step by
        incrementing counter by 1 till counter $\leftarrow$ size -1, where
        integer variable i, counter is set to 0 initially:
        if p[counter] == sym[0]:
                a.  Append [counter, counter + 1] to a1.
        else:
                b.  Append [counter, counter + 1] to b1.
  iii.  Store sym[0] in a char variable, symbol and store states
        whose transitions are missing in dfa into a 2-D list, miss
        and j, k is initialized to 0 initially and create two empty
        strings, prefix and temp_s. Repeat the subsequent step till
        i $\leftarrow$ size, incrementing i by 1 in each step:
        if i $\leftarrow$ size – 2:
                a.  Store empty string in an array, states at
                    location i.
        else:
                b.  Store substring of pattern p starting from
                    index i till the end into states[i].
  iv.   Store i $\leftarrow$ miss[k]. Repeat the subsequent step till j $\leftarrow$ i,
        incrementing j by 1 in each iteration and reset j $\leftarrow$ 0 after
        the last iteration:
                prefix $\leftarrow$ prefix – states[j][0].
                temp_s $\leftarrow$ prefix + symbol.
  v.    Append (temp_s +states[j]) to string_list repeatedly
        incrementing j by 1 each time till j $\leftarrow$ size.
  vi.   Declare an array size_comp. Repeat the subsequent step
        till l $\leftarrow$ size(string_list), incrementing l by 1 in each step
        where l is an integer variable initially 0:
                j $\leftarrow$ string_list[l]
                if(checkPattern(p,j,size)):
                        a.  Append j to size_comp.
  vii.  Find the position of element of smallest length in
        size_comp and store it in minElementPos.
        if symbol == sym[0]:
                a.  Append [i, minElementPos] to a1.
        else:
                b.  Append [i,minElementPos] to b1.
  viii. Repeat thru step 3 till counter $\leftarrow$ size(miss). Increment k
        by 1.
  ix.   Repeat thru step 4 till k $\leftarrow$ size(miss).
End
```

Figure 6: Algorithm to Generate a DFA for a Given Suffix Pattern

C. *Algorithm 3: DFA for Substring Pattern*

It takes the alphabet symbols as input along with the substring pattern and outputs the DFA for any combination of input symbols having the input substring.

```
Begin
  i.    Input the alphabet symbols in an array, sym and input the
        substring pattern in a string, p.
  ii.   Use Algorithm 2 of generating a DFA for a given suffix
        pattern.
  iii.  Declare and initialize a variable i $\leftarrow$ sym[0].
  iv.   Store transition of i for final state to itself.
  v.    i $\leftarrow$ sym[1].
  vi.   Store transition of i for final state to itself.
End
```

Figure 7: Algorithm to Generate a DFA for a Given Substring Pattern

A. *Algorithm 4: To Draw Transition Graph*

It takes transition table as input and output transition graph

```
Begin
  i.    Read transition table, and then extract total
        alphabets (a) and total no. of states (n).
  ii.   If transition table has dead state then set
        flag $\leftarrow$1.
  iii.  Draw n circles and store their center
        coordinates.
  iv.   If flag $\leftarrow$1
  v.    Then: final state will be (n-1)th circle,
        draw encircle in it.
  vi.   Else
  vii.  Then: final state will be nth circle, draw
        encircle in it.
  viii. Draw transitions using transition table,
        using coordinates of circles draw arrows
        between them.
End
```

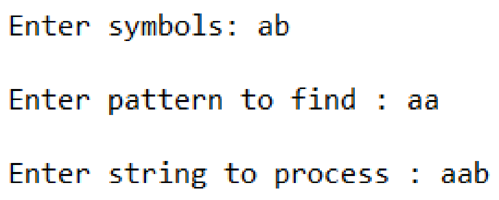
Figure 8: Transition Graph Generation Algorithm

IV. RESULTS AND DISCUSSIONS

In general the proposed algorithms can be implemented in any programming language but with context to this paper the results are obtained by implementing proposed algorithms in python programming language. The results are tested by generating transition table, transition graph, and string processing for various string patters. Snapshot for all these outputs for suffix, prefix and substring patterns for a given string over $\Sigma = \{a.b\}$ are shown in the context below in comparison with JFLAP, which is entirely manual and laborious. User needs to draw the transition diagram manually for the given pattern then only it can perform string processing.. In every output transition table is generated in which '$\rightarrow$' represents initial state and '**' represents final state. Table 2 shows what are the inputs and Table 3 shows outputs by proposed algorithm and JFLAP tool.

A. *Generating DFA for given prefix pattern*

The string “aab” is checked for recognition using transition table constructed in figure 9 and output is shown in figure 10 and 11. The same string “aab” is checked for recognition by a recognizer designed in figure 12 in JFLAP and its output is shown in figure 13.

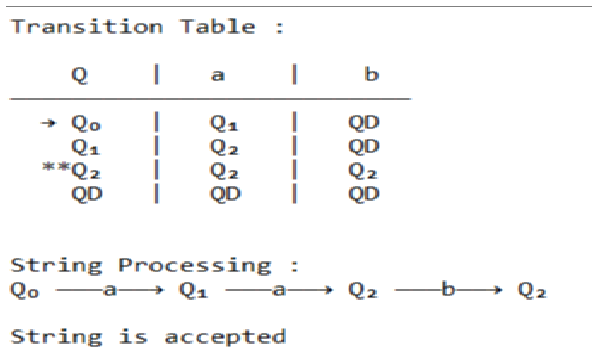


```
Enter symbols: ab

Enter pattern to find : aa

Enter string to process : aab
```

Figure 9: Prefix Pattern and String Input



```
Transition Table :

      Q    |    a    |    b
 -------------------------------
  → Q0     |   Q1    |   QD
    Q1     |   Q2    |   QD
  **Q2     |   Q2    |   Q2
    QD     |   QD    |   QD


String Processing :
Q0 ──a──→ Q1 ──a──→ Q2 ──b──→ Q2

String is accepted
```

Figure 10: Transition Table and String Processing for Given Prefix Pattern

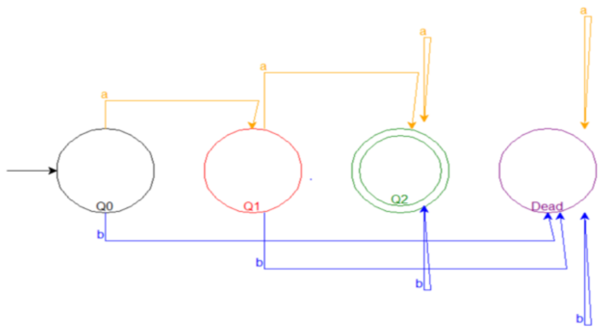


Figure 11: Transition Graph for Given Prefix Pattern

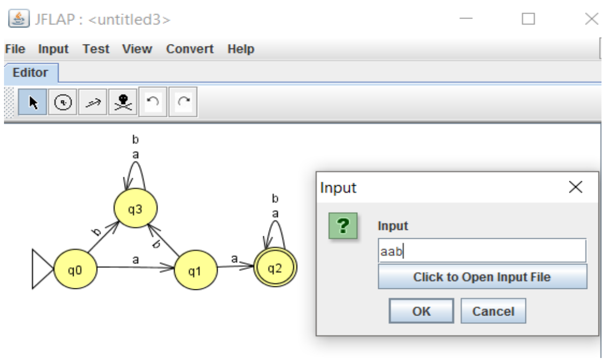
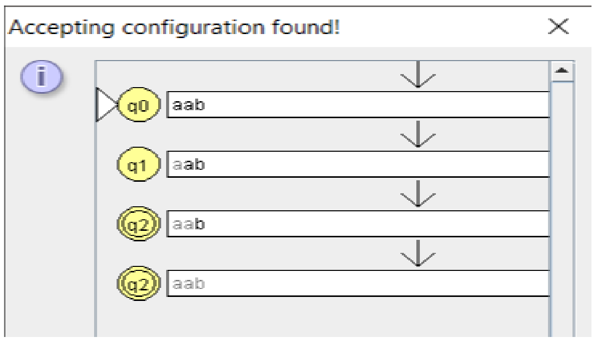


Figure 12: JFLAP Transition Graph for Given Prefix Pattern



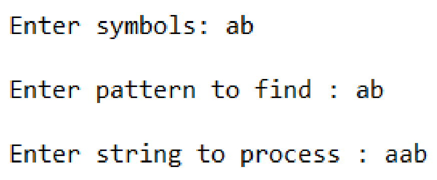
Accepting configuration found!

```
q0  aab
q1  aab
q2  aab
q2  aab
```

Figure 13: JFLAP String Processing for Given Prefix Pattern

B. *Generating DFA for given suffix pattern*

The string “aab” is checked for recognition using transition table constructed in figure 14 and output is shown in figure 15 and 16. The same string “aab” is checked for recognition by a recognizer designed in figure 17 in JFLAP and its output is shown in figure 18.

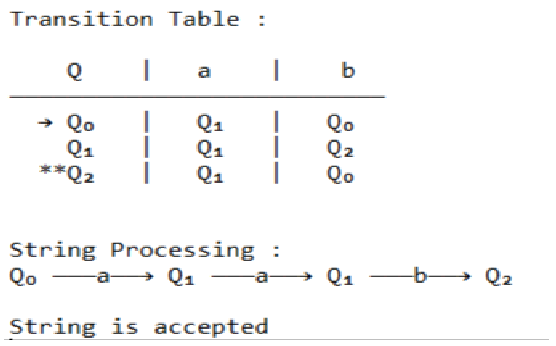


```
Enter symbols: ab

Enter pattern to find : ab

Enter string to process : aab
```

Figure 14: Suffix Pattern and String Input



```
Transition Table :

      Q      |    a     |     b
  ─────────────────────────────────
   → Q₀      |    Q₁    |     Q₀
     Q₁      |    Q₁    |     Q₂
   **Q₂      |    Q₁    |     Q₀


String Processing :
Q₀ ──a──→ Q₁ ──a──→ Q₁ ──b──→ Q₂

String is accepted
```

Figure 15: Transition Table and String Processing for Given Suffix Pattern

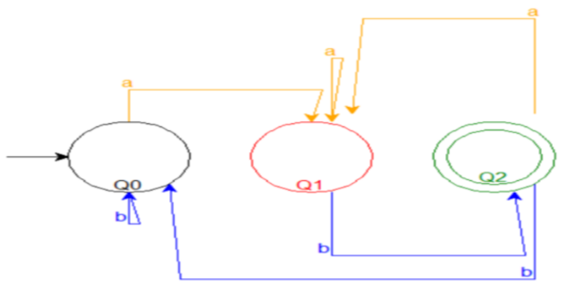


Figure 16: Transition Graph for Given Suffix Pattern

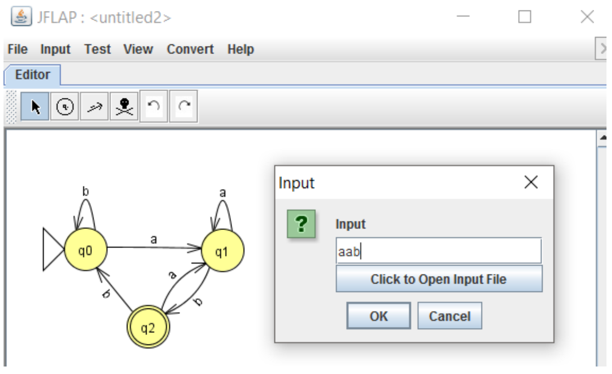
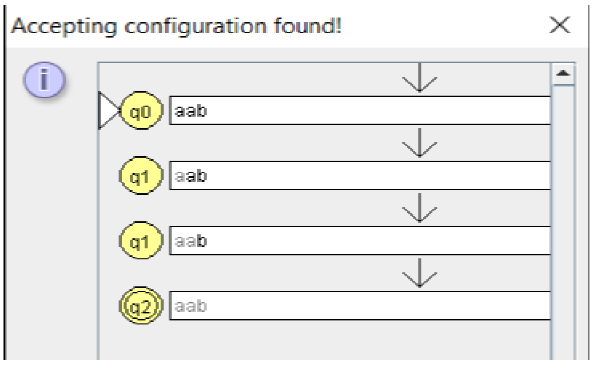


Figure 17: JFLAP Transition Graph for Given Suffix Pattern



Accepting configuration found!

q0 aab

q1 aab

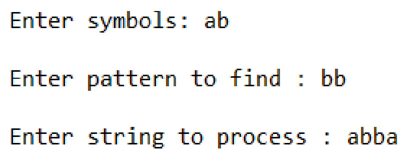
q1 aab

q2 aab

Figure 18: JFLAP String Processing for Given Suffix Pattern

C. Generating DFA for given substring pattern

The string "abba" is checked for recognition using transition table constructed in figure 14 and output is shown in figure 15 and 16. The same string "abba" is checked for recognition by a recognizer designed in figure 17 in JFLAP and its output is shown in figure 18.

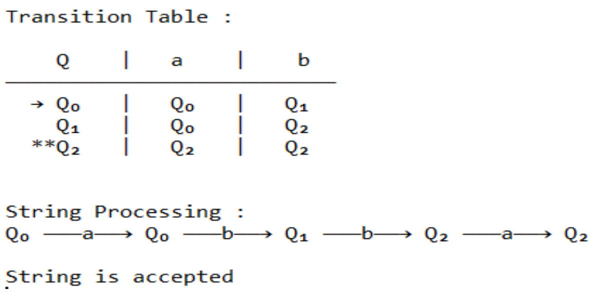


```
Enter symbols: ab

Enter pattern to find : bb

Enter string to process : abba
```

Figure 19: Substring Pattern and String Input



```
Transition Table :

      Q      |    a     |     b
  ─────────────────────────────────
   → Q₀      |    Q₀    |     Q₁
     Q₁      |    Q₀    |     Q₂
   **Q₂      |    Q₂    |     Q₂


String Processing :
Q₀ ──a──→ Q₀ ──b──→ Q₁ ──b──→ Q₂ ──a──→ Q₂

String is accepted
```

Figure 20: Transition Table for Given Substring Pattern

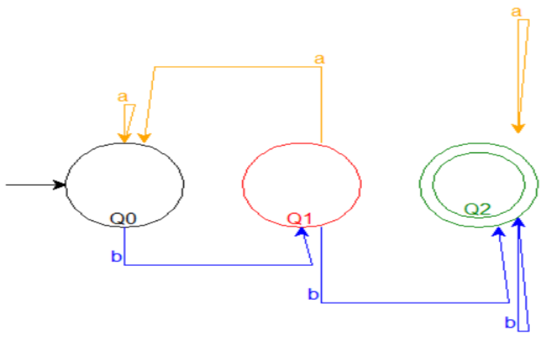


Figure 21: Transition Graph for Given Substring Pattern

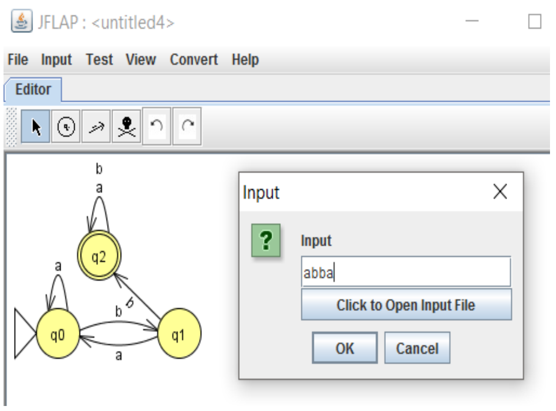


Figure 22: JFLAP Transition Graph for Given Substring Pattern

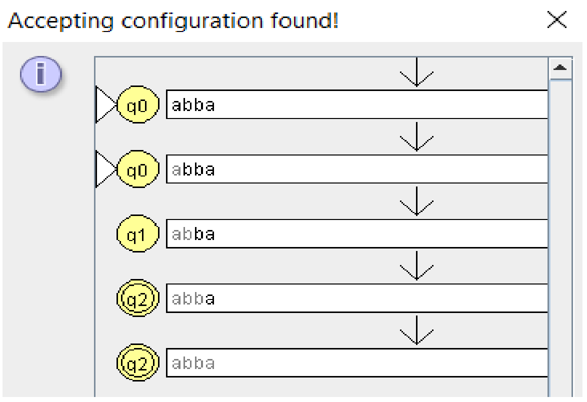


Figure 23: JFLAP String Processing for Given Substring Pattern

The results of string recognition by both transition table and JFLAP are same as both are showing the same state transitions and string acceptance. However, the transition graphs are constructed in JFLAP tool manually as shown in figure no. 12, 17 and 22, whereas our proposed algorithm implemented in Python language generates the same automatically as shown in figure no.11, 16, and 21 making it effortless and time-efficient.

V. CONCLUSION AND FUTURE SCOPE

Deterministic Finite Automata is a model of computation in which for each state an input symbol there is single transition. DFA finds application in the field of compiler design. Also, it gives the acceptance or rejections of the user entered strings according to DFA. In present paper, the algorithms for DFA design are proposed for suffix, prefix and substring patterns. Implementation is done in python and is compared with the JFLAP tool. Results reveal that the performance of auto generation of DFA implementation gives the better performance than JFLAP tool. Design of DFA automatically handles all types of validations and follows the rules of DFA. Design of DFA incorporates the following features: transition graph and transition table of DFA, which is helpful for better understanding. In addition, it gives the acceptance or rejections of strings after string processing DFA. In future, the algorithm can be simplified and its time and space complexity can be taken into consideration to make the algorithm computationally more efficient. Algorithms for other language string types can also be proposed.

REFERENCES

[1] Hopcroft J.E and Ullman J.D, “Introduction to Automata Theory, Languages and Computation”, Addison – Wesley, pp. 37-53, 1979.

[2] Ulman, J.D. (1972), “Applications of language theory to compiler design”, proceedings of thr May 16-18, 1972, spring joint computer conference, pp. 235-242, 1972.

[3] Gribko E. "Applications of Deterministic Finite Automata" ECS 120 UC Davis, spring 2013.

[4] Murugesan NG., “Principles of Automata theory and Computation. Sahithi Publications”; 2004.

[5] Parekh, R. G., Honavar, V. G., "Learning DFA from Simple Examples" Journal of Machine Learning, Vol. 44, Issue 1-2, pp. 9-35, 2001.

[6] Murugesan N, and Samyukthavarthini B, “A Study on Various types of Automata”, 2013.

[7] Ejendibia P., Baridam B. B., “String Searching with DFA based Algorithm”, International Journal of Applied Information systems, Vol. 9, No. 8, 2015.

[8] BabuKaruppiah A., Rajaram S., "Deterministic Finite Automata for pattern matching in FGPA for intrusion detection" International Conference on Computer, Communication and Electrical Technology, pp. 167-170, 2011

[9] Shenoy V., Aparanji U., Sripradha K., Kumar V., "Generating DFA Construction Problems Automatically" International Journal of Computer Trends and Technology, Vol. 4, Issue 4, pp.32-37,2013